（续表）

十六进制	十进制	二进制
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

因为 1 个十六进制位刚好对应 4 个二进制位，所以要把二进制数表示为十六进制数，可以先从数字的右边开始，提取包含 4 个二进制位的组，再为每个组写出对应的十六进制位。看看下面的二进制数：

1111 0101 1011 1001 1110 0001

如果把每组的 4 个二进制位替换为对应的十六进制位，就得到：

F5B9E1

得到的 6 个十六进制数字对应于 6 组 4 个二进制位。为了说明这是正确的，再次使用与十进制类似的方式，把这个数字从十六进制转换为十进制：

F5B9E1 是：

$$15 \times (16 \times 16 \times 16 \times 16 \times 16) +$$
$$5 \times (16 \times 16 \times 16 \times 16) +$$
$$11 \times (16 \times 16 \times 16) +$$
$$9 \times (16 \times 16) \times 14 \times (16) + 1$$

得到：

$$15\ 728\ 640 + 327\ 680 + 45\ 056 + 2\ 304 + 224 + 1$$

和正好等于把最初的二进制数转换为十进制数后的结果。

B.3　八进制数

第 2 章提到，可以定义八进制的整数字面量，即基数为 8 的整数。八进制以及把数字表示为基于 64 均由瑞典的 Charles XII 于 1717 年发明，但这不是它们出现在 Java 中的原因。产生八进制表示法的原因是：以前计算机把二进制整数保存在是 3 位二进制数字倍数的单元中，20 世纪 60 年代初期古老的 IBM 7090 使用 36 位存储整数，这就是使用 12 个八进制数字指定 36 位二进制值的典型示例。那时的一些编程语言把二进制值写为八进制，因为这样可以节省空间。随着编程语言逐渐演变为更大、更好、更强大的语言，八进制表示法到今天仍在使用。八进制数字仍在 C、C++和现在

的 Java 中使用，尽管它们的作用不大。

八进制的数字位从 0 到 7，计算类似于十六进制，但基数是 8 而不是 16。八进制字面量很容易与十进制整数混淆。在 Java 中，它们唯一的区别是八进制字面量有前导 0。除非绝对必须这么做，否则最好避免使用八进制字面量。

B.4 负的二进制数

需要理解的二进制算术的另一个方面是如何表示负数。前面假定所有数字都是正的。乐观主义者的看法是：杯子仍是半满的，负数是永远不能避免的。悲观主义者的看法是：杯子是半空的。如何表示负数？我们只有二进制数字，但它们必须采用某种方式表示负数。

对于允许包含负值的数字(称为带符号的数字)而言，必须先确定固定长度(换言之，指定二进制数的位数)，再把最左边的二进制数指定为符号位。使数字的位数固定，可以明确指定哪个位是符号位，而其他位是数字位。一个位就足以表示数字的符号，因为数字要么是正的——符号位是 0，要么是负的——符号位是 1。

当然，一些数字有 8 位，一些数字有 16 位，无论位数有多少，只要知道每个数字有多少位即可。如果符号位是 0，数字就是正的；如果符号位是 1，数字就是负的。这似乎解决了问题，但没有完全解决。如果使用二进制把−8 和+12 相加，就会得到结果+4。如果过分简单地执行这个计算，只把正数的符号位加 1，使之变成负数，再用传统的方式对剩下的位执行算术计算，结果就不正确：

12 在二进制中是	0000 1100
−8 在二进制中是	1000 1000

因为+8 是 0000 1000，所以−8 的二进制表示与+8 相同，但最左边的位是 1。现在把它们相加，得到：

12 + (−8)是	1001 0100

根据规则，1001 0100 的十进制值是−20，这根本不是我们希望的结果。肯定不是+4，因为+4 的二进制表示是 0000 0100。读者可能以为：“不能把符号看作另一个数字位”。但计算机在处理二进制数时，只能这么做，因为计算机是机器，不会用其他方式处理。所以为了使相加操作正确执行，而不管操作数的符号如何，就需要为负数使用另一种标识符。不管操作数的符号如何，算术操作都应能正确执行，试试从+4 中减去+12，应得到结果−8：

+4 是	0000 0100
减去+12	0000 1100
得到	1111 1000

对于从右数的第 4 个数字开始，都必须借 1 才能求出结果，这类似于十进制中的减法。结果是 8，但看上去不像。在二进制中，对+4 加上+12 或+15，结果是正确的。这称为二进制负数的补 2 表示法。

现在需要读者对这种表示法有信心，但这里不说明工作原理，只是介绍如何从正值中构建负数的补 2 表示法，而且这是有效的，读者可以自己证明。下面回到第一个例子，需要−8 的补 2 表示法。

−8 的二进制是：

0000 1000

现在翻转每个二进制位——如果某个二进制位是 1，就使之变成 0，反之亦然：

1111 0111

这称为补 1 形式，对之加上 1，就得到补 2 形式：

	1111 0111
对之加上 1	<u>0000 0001</u>
得到	1111 1000

这看上去类似于从+4 中减去+12 后得到的−8 的表示法。为了进一步加以确定，把−8 和+12 相加：

+12 是	0000 1100
−8 的新版本是	<u>1111 1000</u>
得到	0000 0100

答案是 4——真神奇，这是正确的！进位操作一直传递到最左边的 1，把它们都变回 0。最后的进位被舍弃了，但不用担心，它是在前面用来得到−8 的减法中借来的。实际上，我们做了如下隐含的假设：符号位 1 或 0 永远在最左边。如果自己试着计算几个例子，就会发现这种方法总是有效。而且这使计算机的算术运算变得非常简单、快速。

B.5 浮点数

我们常常要处理非常大的数字，例如宇宙中质子的数量约需要 79 个十进制数字位才能表示。显然，在许多情况下，需要的十进制数字都超过了 4 字节的二进制数。同样还有许多非常小的数，例如汽车销售员开着 1999 款本田雅阁车行驶了 387 604 英里，而且没有使用钥匙启动汽车，因为钥匙丢在路上了。于是他在听取顾客的议价时，需要几分钟的时间来确定。处理这两种数字的机制是浮点数。

浮点数是指使用小数点后跟固定位数的数字与 10 的幂的乘积，进而表示希望的数字。演示这种表示法相比解释起来更容易，所以下面举一些例子。对于使用十进制表示的数字 365，浮点数形式为：

0.365E03

其中 E 表示 exponent(指数)，是 10 的幂，0.365(尾数)乘以这个幂才能得到需要的值：

$$0.365 \times (10 \times 10 \times 10)$$

显然结果是 365。

下面再看一个小一些的数：

.365E−04

表示 $.365 \times 10^{-4} = .0000365$，是指汽车销售员接受现金、打开"Closed"信号所需的时间(以分钟为单位)。

浮点数中尾数的位数称为精度，取决于所用浮点数的类型。Java 类型 float 约精确到小数点后 7 位，而 double 类型约精确到小数点后 17 位。精确到小数点后多少位是大约的数，因为尾数是二进制而不是十进制，二进制和十进制之间没有准确的映射。

假定有一个很大的数 2 134 311 179，如何将之表示为浮点数？float 类型的数字的十进制表示是：

0.2134311E10

它们不完全相同，因为最后 3 位数字被舍弃了，所以最初的数字约等于 2 134 311 000。如果要处理的数字范围非常大，一般是从 10^{-38} 到 10^{+38} 的正数或负数以及从非常小的 10^{-308} 到非常大的 10^{+308}，那么这将是很小的代价。可以看出，将它们称为浮点数的明显原因是：根据指数值的不同，小数点是浮动的。

除了为确保精确而出现的精度限制之外，还有一个方面需要考虑。在对数量级完全不同的数字执行加法或减法时，需要特别小心。一个简单的例子就可以说明可能出现的问题。先对.365E－3 和.365E+7 求和，把它们写为小数值的和：

.000365 + 3,650,000

结果是：

3 650 000.000365

再转换回精度为 7 位的浮点数：

.3650000E+7

读者可能还没有发现什么问题，问题在于精度只有 7 位。大数的 7 位没有受到小数的任何位的影响，因为小数的所有位都在这 7 位的右边。另外，在两个数字接近相等时也要小心。如果计算这两个数字之差，结果就可能只精确到 1 或 2 位。在这种情况下，很容易出现用完全是垃圾的数字来计算的情形。

使用浮点数的最后一个要点是：许多十进制小数不能准确地表示为二进制浮点数。例如，十进制的 0.2 不能准确地表示为二进制浮点数。这表示在处理这些值时，从一开始这些值就有微小的错误。影响之一是计算 100 个 0.2 的和不会得到 20。如果在 Java 中实验一下，结果将是 20.000004，略大于希望的结果。

因此，尽管浮点数是在程序中表示范围非常大的数值的一种强大方式，但必须注意局限性。如果对可以处理的数值范围感兴趣，通常可以执行需要的计算来避免上述问题。换言之，如果在使用浮点数时知道使用局限，就可以避免它们。

235个实例、4个项目实战案例，全面解读.NET 4.0数据库开发技术
涵盖C# 4.0、SQL Server 2008、ADO.NET 4.0及LINQ等最新技术

精通 C#与.NET 4.0 数据库开发

——基础、数据库核心技术、项目实战

（43.5小时多媒体教学视频）

秦婧 石叶平 等编著

◎ 从.NET框架入手，详细介绍C# 4.0基础及其关键开发技术
◎ 全面剖析SQL Server 2008、ADO.NET及LINQ等数据库开发技术
◎ 从数据的查询、连接、缓存、优化等角度展现C#开发的优势
◎ 注重实战，详细介绍4个有针对性的项目案例供读者实战演练
◎ 每章的重点内容都配有多媒体教学视频，学习起来高效、直观



清华大学出版社

精通

C#与.NET 4.0 数据库开发

——基础、数据库核心技术、项目实战

秦婧 石叶平 等编著

清华大学出版社

北 京

内 容 简 介

本书从实战出发，全面介绍了微软.NET 4.0 平台所涉及的 SQL Server 2008、ADO.NET 和 LINQ 等数据库开发技术。书中提供了大量实例，并提供了 4 个有针对性的项目案例供读者实战演练。

本书附带 1 张 DVD 光盘，内容为本书涉及的源代码和配套的教学视频，另外还赠送了 C#、ASP.NET 和 SQL Server 入门教学视频等其他学习资料。

本书共分 6 篇。第 1 篇介绍了.NET 框架、C#基本语法、C#类和接口、C#高级特性；第 2 篇介绍了 Windows 窗体程序、多文档 Windows 窗体程序、.NET 类库开发、ASP.NET 网页开发；第 3 篇介绍了 SQL Server 2008 入门、Transact-SQL 语言；第 4 篇介绍了使用 ADO.NET 表示数据库、使用 ADO.NET 访问数据库、使用.NET 数据绑定；第 5 篇介绍了 LINQ 查询基础、LINQ to DataSet、LINQ to SQL、LINQ to XML；第 6 篇介绍了 4 个有针对性的项目案例：ATM 交易管理系统、进销存管理信息系统、宾馆管理信息系统、ME 校友录，这些案例对提高读者的数据库开发水平有很大帮助。

本书内容丰富，重点突出，适合 C#语言学习人员、.NET 程序员和数据库开发人员阅读，尤其适合想提高实际开发水平的人员阅读。另外，本书实用性强，很适合相关培训学校的学员作为教材使用。

图书在版编目（CIP）数据

精通 C#与.NET 4.0 数据库开发：基础、数据库核心技术、项目实战 / 秦靖等编著. —北京：清华大学出版社，2011.1（2017.1 重印）

ISBN 978-7-302-24121-8

Ⅰ. ①精…　Ⅱ. ①秦…　Ⅲ. ①C 语言 – 程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 232263 号

责任编辑： 夏兆彦
责任校对： 徐俊伟
责任印制： 何　芉

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 虎彩印艺股份有限公司
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 35　　**字　　数：** 874 千字
（附 DVD 1 张）
版　　次： 2011 年 1 月第 1 版　　**印　　次：** 2017 年 1 月第 5 次印刷
印　　数： 8601～9250
定　　价： 69.00 元

产品编号：040276-01

前　言

为什么要写这本书？

C#语言为什么会越来越流行呢？这归功于微软的大力支持。微软在新推出的 Visual Studio 2010 集成开发工具中，照例用 C#作为主要开发语言，并提供了完善的.NET 底层类库支持。现在的应用程序种类越来越多，C#就可以支持各种应用程序的开发，如 Windows 窗体应用程序、类库、Web 网络应用程序等。这样我们就不用疲于更换各种语言。凡是大型应用，必定会涉及数据操作，数据可大可小，可以有 XML 数据库，可以有 SQL Server 数据库，还可以有一些小型数组。本书就是为了处理数据而推出的一本专用于项目数据操作的书，目的旨在让读者熟悉 C#语言的基础上，还能熟悉语言的数据处理能力。

为了让读者能够层层递进地学习，本书前面先介绍了 C#的语法基础和面向对象开发的一些特点，然后介绍了 T-SQL 语句处理、ADO.NET 数据处理、LINQ 数据查询、XML 数据处理等数据库开发技术，最后给出了 4 个有针对性的项目案例，以提高读者的实战水平。本书讲解采用理论结合实例的形式，务求看了必会，会了必能动手。

本书有何特色？

1．紧跟行业发展，关注最新技术

本书针对微软最新的 Visual Studio 2010 开发平台而写，所涉及的内容都是目前的最新版本和技术，如 C# 4.0、ASP.NET 4.0、ADO.NET 4.0、SQL Server 2008 等均为最新版本。书中对微软最新的 LINQ 数据查询技术也做了重点介绍。

2．配超值DVD视频教学光盘

本书配带 1 张非常超值的 DVD 光盘，内容如下：

- 本书配套多媒体教学视频；
- 本书所涉及的源代码；
- C#入门教学视频（免费赠送）；
- ASP.NET 入门教学视频（免费赠送）；
- SQL Server 入门教学视频（免费赠送）；
- 其他学习资料（免费赠送）。

3．讲解循序渐进，重点突出

本书首先介绍了 C#语言的基础知识，然后重点介绍了.NET 平台的数据库开发技术，

最后基于实战介绍了 4 个数据库项目案例。

4．实例丰富，易学易用

本书讲解时理论结合实践，并穿插了大量的典型实例帮助读者理解书中的内容，对于一些容易在程序中出错的技术点和难点也做了专门讲解，读者掌握起来非常容易。

5．精选项目案例，实用性强

本书精选了 4 个数据库项目开发案例，这 4 个案例分别基于书中所讲解的 T-SQL 语句处理、ADO.NET 数据处理、LINQ 数据查询、XML 数据处理等内容，非常有针对性，可以大大提升读者的数据库开发能力。

本书内容及知识体系

第1篇　C# 4.0语言基础（第1～4章）

本篇主要介绍了 C# 4.0 语言基础，包括.NET 的底层框架和面向对象开发等知识。最后还介绍了泛型、委托等 C#的高级特性。

第2篇　开发应用程序（第5～8章）

本篇重点介绍了 C#在 Windows 窗体程序、多文档 Windows 窗体程序、.NET 类库开发、ASP.NET 网页开发等开发领域的应用。

第3篇　SQL Server 2008基础（第9～10章）

本篇主要介绍了微软最新的数据库 SQL Server 2008 的一些常用操作，另外还介绍了标准数据库查询语句 SQL 的应用。

第4篇　ADO.NET操作数据库（第11～13章）

本篇重点介绍了使用 ADO.NET 表示数据库和访问数据库，另外还专门介绍了.NET 数据绑定的相关知识。ADO.NET 是一个类库，它提供了一系列类方便开发人员调用数据库。有了 ADO.NET，数据库应用程序的编写可以节约大量代码。

第5篇　LINQ查询开发（第14～17章）

本篇重点介绍了微软最新推出的 LINQ 数据查询的相关技术。LINQ 是.NET 平台数据查询的后起之秀，它支持各种数据类型，如通过 LINQ to XML 可以处理 XML 数据，它封装了数据查询和各种类型数据操作的一些简便方法，目的是提高数据处理能力。

第6篇　项目实战（第18～21章）

本篇重点介绍了 4 个数据库项目案例的实现，分别针对.NET 平台的各种数据库开发技术，如用 T-SQL 开发数据库应用系统，用 ADO.NET 处理应用程序中的数据，用 LINQ to SQL 处理数据实体类，用 XML 提供应用程序数据。

适合阅读本书的读者

- C#语言初学者；
- 有 C#语言基础，想进一步学习项目开发的人员；
- C#与.NET 数据库开发人员；
- 想了解.NET 平台最新技术的人员；
- 大中专院校的学生；
- 相关培训学校的学员。

本书作者及编委会成员

本书由秦婧、石叶平主笔编写。其他参与编写的人员有班志杰、陈旭、陈永俊、陈争光、戴建华、方文票、冯玉荣、高姗姗、巩宁来、谷世江、胡其吐、黄飞龙、蒋晓捷、李德明、李显亮、李志勇、刘雁征、吕小波、马东、孟庆海、唐勇、王浩、王玲玉、王志娟、武娜、徐晓娟、闫树丰、杨朝宇、翟闯等。在此表示感谢！

本书编委会成员有欧振旭、陈杰、陈冠军、项宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。

编著者

目　　录

第 1 篇　C# 4.0 语言基础

第 2 篇　开发应用程序

第 3 篇　SQL Server 2008 基础

第 4 篇 ADO.NET 操作数据库

第 5 篇　LINQ 查询开发

第 6 篇 项目实战

第 1 篇　C# 4.0 语言基础

第 1 章　了解.NET 框架

随着计算机的广泛应用和软件技术的发展，不同的操作系统、开发平台、运行框架、开发技术的不兼容性日益突出，给开发人员带来很大麻烦，甚至已经严重影响到软件技术的发展。微软公司借助.NET 框架将 Windows 下各种应用程序开发有机地集成起来，为开发人员提供统一的开发接口和类库。本节将介绍.NET 的基本知识。

1.1　.NET 的产生和发展

在众多 Windows 开发语言相互不兼容的情况下，微软推出.NET 框架及其开发环境，将 Windows 应用程序的开发带进新的历史时代。本节将介绍.NET 的产生及其发展历程。

1.1.1　.NET 的产生

计算机是现代社会必不可少的科技产品，它被广泛应用在人们的日常工作和生活中，给人们的生活和工作带来巨大的影响。从日历闹钟到音乐电影，从本地阅读到网上冲浪，从办公文档到公司管理，软件出现在人们工作和生活的方方面面。计算机软件的发展是一个从小到大，从简单到复杂的不断改进和统一的过程。无论是软件的开发语言还是开发流程都有了很大进步，本书的重点是开发语言，所以不会过多地讨论开发流程方面的知识。

目前，Windows 操作系统在国内外都占有相当大的客户群体，几乎处于不可动摇的地位。所以软件开发项目尤其是中小型项目更多是面向 Windows 的。然而在计算机软件发展过程中，不断有各种各样新的技术产生，也有落后的技术被淘汰，各种技术很难统一集成到一起。比如，在 Windows 下进行软件开发，就有多种不同的相互不兼容的技术：

- 在图形图像开发方面，有 GDI、DirectX、OpenGL 等 3 种模式，且互不兼容。
- 在数据库操作方面，有 ADO、DAO、RDO、ODBC 等 4 种模式，且互不兼容。
- 在网站开发技术方面，有 ASP、JSP 等 2 种语言，且互不兼容。
- Windows 本身有服务器版、专业版、Home 版等多种不完全兼容的版本。

这一系列的问题，都给软件设计和开发带来很大的困难和冗余工作。虽然 COM 组件利用面向对象思想，试图通过接口的方式来达到更多的模块重用和统一接口，但是它在版本管理、组件部署、组件继承的方面的缺陷使得它逐渐退出历史舞台。此外，网站开发和桌面程序开发在模式上存在的巨大差异，也大大降低了组件的重用率。

为了解决这些问题，微软推出了一套新的解决方案——Microsoft .NET Framework。.NET 框架是一个灵活、稳定的能够运行 Web 服务和 Windows 程序的 Windows 内置组件。它既是软件的运行环境，又是软件开发和存在的基础。.NET 框架具有以下特性：

- .NET 框架将 Windows 操作系统底层的 API 进行封装，并为不同 Windows 提供了统一的应用层接口，从而消除了 Windows 操作系统带来的不一致性。
- .NET 框架在设计上具有 COM 组件的统一性，同时，它还提供了用户认证信息管理、应用程序版本管理和应用程序部署等。
- .NET 框架用面向对象思想，围绕继承这一概念设计，力求代码和组件重用。它所提供的所有类库是一个个相对独立的模块，可以广泛应用在软件开发的各领域。
- .NET 框架支持多种开发语言，它通过公共语言规范（CLS）将 VB.NET、VC++.NET、C#、Visual J#等多种语言统一起来。只要是符合 CLS 的开发语言都可以被.NET 支持。
- .NET 另外一个重大改进就在于网页开发的改进，它采用将网页分成前台网页和后台代码的前后台开发方式。将页面开发和应用逻辑开发完全分离，大大提高了网页开发效率，及组件和代码的重用。

1.1.2　.NET 的发展

.NET 的全名叫.NET Framework（.NET 框架）。早在 2001 年，微软就发布了.NET 的第一个版本之后，经过了 10 年的不断改进和努力，至今被广大开发人员所接受和认可，共经历了 6 个不同的版本。

- .NET Framework 1.0：2001 年正式推出，也是.NET 的第一个版本，它以 SDK 的形式存在，与 Visual Studio 2002 集成发布。
- .NET Framework 1.1：2003 年第一次发布，并经过多次修正，在 2004 年基本完善，这是.NET 历史上非常重要的一个版本，它和 Visual Studio 2003 一起发布。.NET 1.1 版本提供了 ASP.NET 组件，而且原有的控件都作为独立的.NET 类库提供，将框架和类库完全分离。此外，还将.NET 分成标准版和精简版，其中精简版主要适用于移动设备的开发上，这使得.NET 的应用更加广泛。
- .NET Framework 2.0：2004 年第一次发布，同样经过了一年的改进和完善，在 2005 年正式稳定。该版本也是.NET 历史上非常流行的一个版本，它是伴随着 Visual Studio 2005 和 SQL Server 2005 发布，所以该版本提供了大量与 SQL Server 集成的开发接口，可以实现安全而高效的数据库访问。.NET 2.0 也在 ASP.NET 方面进行了很大改进，支持主题和皮肤等个性化方面的开发。
- .NET Framework 3.0：.NET 3.0 发布于 2006 年，它在内核上与早期的.NET 版本有很大区别，但单从接口来看，完全是向后兼容。.NET 3.0 完全可以说是 Windows Vista 操作系统的很大一部分，所以它是 Vista 内置支持的.NET 类库，这也为.NET 的进一步发展做出很大贡献。.NET 3.0 主要提供了 Windows 表示层类库（WPF）、Windows 通信类库（WCF）、Windows 工作流类库（WF）及 Windows 数字标识（WCS）4 大基本类库，将.NET 开发推向新的历史阶段。WPF 让 Windows 应用程序用户界面布局偏向于 Web 页面，同时也支持前后台开发模式，用户界面非常高效而美观。WCF 使得基于多种通信协议的开发变得更加统一。
- .NET Framework 3.5：.NET 3.5 发布于 2008 年，更多是对.NET 3.0 的扩充和完善，提供了更加丰富的.NET 类库，提供了 LINQ 组件，通过 LINQ 组件可以非常容易

地实现内存数据的查、数据库查询，以及 XML（Extensible Markup Language，可扩展标记语言）等集合类数据的查询。

- .NET Framework 4.0：.NET 4.0 发布于 2010 年，完善了当初.NET 3.5 版本的一些功能，并没有太大的变化。

在.NET 的众多版本中，.NET 1.0 和.NET 1.1 是完全独立的版本，它们不能与后面的版本兼容，即通过.NET 1.0 或.NET 1.1 开发的应用程序不能运行在后期的.NET 框架上。.NET 2.0、.NET 3.0、.NET 3.5 和.NET 4.0 则是向后兼容的，即.NET 2.0 开发的应用程序可以运行在.NET 3.0、.NET 3.5 和.NET 4.0 环境中，而.NET 3.0 开发的应用程序可以运行在.NET 3.5 和.NET 4.0 环境中。

随着.NET 的发展和演化，.NET 开发所需要的编程语言也在不断完善和增强，C#开发语言作为.NET 的主力开发语言之一，它也从最初简单的 C#1.0 版本，逐步发展到现在的 C#4.0。

1.2　公共语言运行库

平台无关性是.NET 框架的重要特性之一，公共语言运行库则是实现这一目标的核心组件，通过公共语言规范定义统一的.NET 框架开发语言都必须遵守的规则。本节将介绍.NET 公共语言运行库相关知识。

1.2.1　公共语言规范——CLS

公共语言运行库是.NET 框架的最核心组件，提供应用程序最基本的运行环境。公共运行库通过定义公共语言规范（CLS）实现.NET 的平台无关性，以及跨语言编程。所有托管代码都应该遵守通用类型系统（CTS）和公共语言规范（CLS），公共语言规范则定义了所有应用程序都需要的最小的语言功能集合，这样任何支持编写 CLS 的开发语言所编写的代码都可以在.NET 框架下执行。

公共语言规范致力于定义一套完整的足够通用的面向对象语言规范，所以它既要足够详细，从而可以实现更多高级语言功能，又需要足够抽象，使得尽可能多的开发语言可以满足它。下面列出.NET 公共语言规范定义的主要内容。

- 命名规则：CLS 规定，所有类型、成员等在独自的命名空间下必须具有唯一的名称，而且名称不能只是大、小写不同。同样名称不能使用语言关键字。
- 数据类型：CLS 规定，作为面向对象开发语言，除了支持基本的基元类型（如 char、string、int 等）之外，还必须支持数组和枚举。另外，还必须支持类和接口，在类成员的可见性上必须支持私有、继承、公开 3 种。
- 类成员：CLS 规定，类必须支持构造函数、属性、方法、字段和事件；必须支持类成员的重载和覆盖，同时还支持类的继承和向上转化。另外，类对象必须是引用。
- 接口成员：CLS 规定，接口成员不能具有访问性，可以包括字段、属性、方法和事件。

- 异常处理：CLS 规定，必须支持异常处理，而且异常可以继承，从而实现自定义异常。
- 事件支持：CLS 规定，事件必须可以动态发布和订阅，而且事件必须具有唯一名称。
- 泛型支持：CLS 规定，泛型名称必须明确包含泛型的具体类型参数，而且可以支持对类型进行约束。泛型的参数类型必须也满足 CLS 规范。

由此可见，公共语言规范定义了面向对象编程语言所必须支持的语言特性，包括类、接口、事件、异常、命名空间、程序集等。在.NET 中，目前有 C#和 VB.NET 是完全符合 CLS 规范的开发语言，这两门语言都是由微软研发，目前也是主流的.NET 开发语言，本书的所有实例都是通过 C#来实现的。

注意：由于篇幅限制，这里只列出了 CLS 规范很少的一部分，关于 CLS 规范更多更详细的内容，可以参考 MSDN 或微软官方网站。

1.2.2　中间语言——MSIL

在.NET 框架下，公共语言运行库和公共语言规范是.NET 实现跨语言和跨平台的基础，然而对于各种各样符合 CLS 的高级开发语言，它们又如何统一到一起呢？这就需要使用到微软中间语言（MSIL），这是一种符合 CLS 且风格颇似汇编的中级语言。之所以说它是中级语言，是因为它并非像真正的汇编语言那样和硬件指令紧密结合，而是由一些模拟的高效的低级指令组成。

在.NET 环境中，通过.NET 开发语言（如 C#、VB.NET 等）开发的应用程序，需要经过如图 1-1 所示的步骤，才能成为最终在计算机上执行的程序。

（1）首先，通过源代码编辑器开发对应的高级语言代码，可以用任何的文本编辑器，比如记事本、写字板等。但是，最友好高效的当然是 Visual Studio 2008，它对 VB.NET 和 C#具有完整而且超强的支持。

（2）然后，通过编译器编译源代码，并生成中间文件。不同高级语言根据自身语言特点及语法的不同，都具有各自专用的编译器。VB.NET 和 C#就拥有各自不同的编译器。

（3）接下来，通过链接器将编译器输出链接成中间语言代码。不同高级语言具有不同的链接器。但是具有相同功能的不同语言开发的代码段，最终产生的中间语言代码却是很相似的。Visual Studio 2008 可以通过一个菜单命令，自动完成编译和链接。

（4）然后，当运行应用程序时，.NET 框架提供的实时编译器将 IL 代码编译成本地机器可以执行的二进制机器代码，此过程中会对代码的类型安全，版本认证等信息进行检查。

（5）最后，机器代码才在目标计算机上运行。

通过上面的步骤可以很清楚地看出，.NET 实现多语言互操作和跨平台的基础就是公共语言运行库与公共语言规范。任何执行于.NET 框架上的高级语言都必须提供对应的编译器和链接器，以便将对应的源代码生成为 MSIL 代码。此外，.NET 应用程序实际保存的是 MSIL 中间代码，而不是直接运行于操作系统上的二进制机器代码。

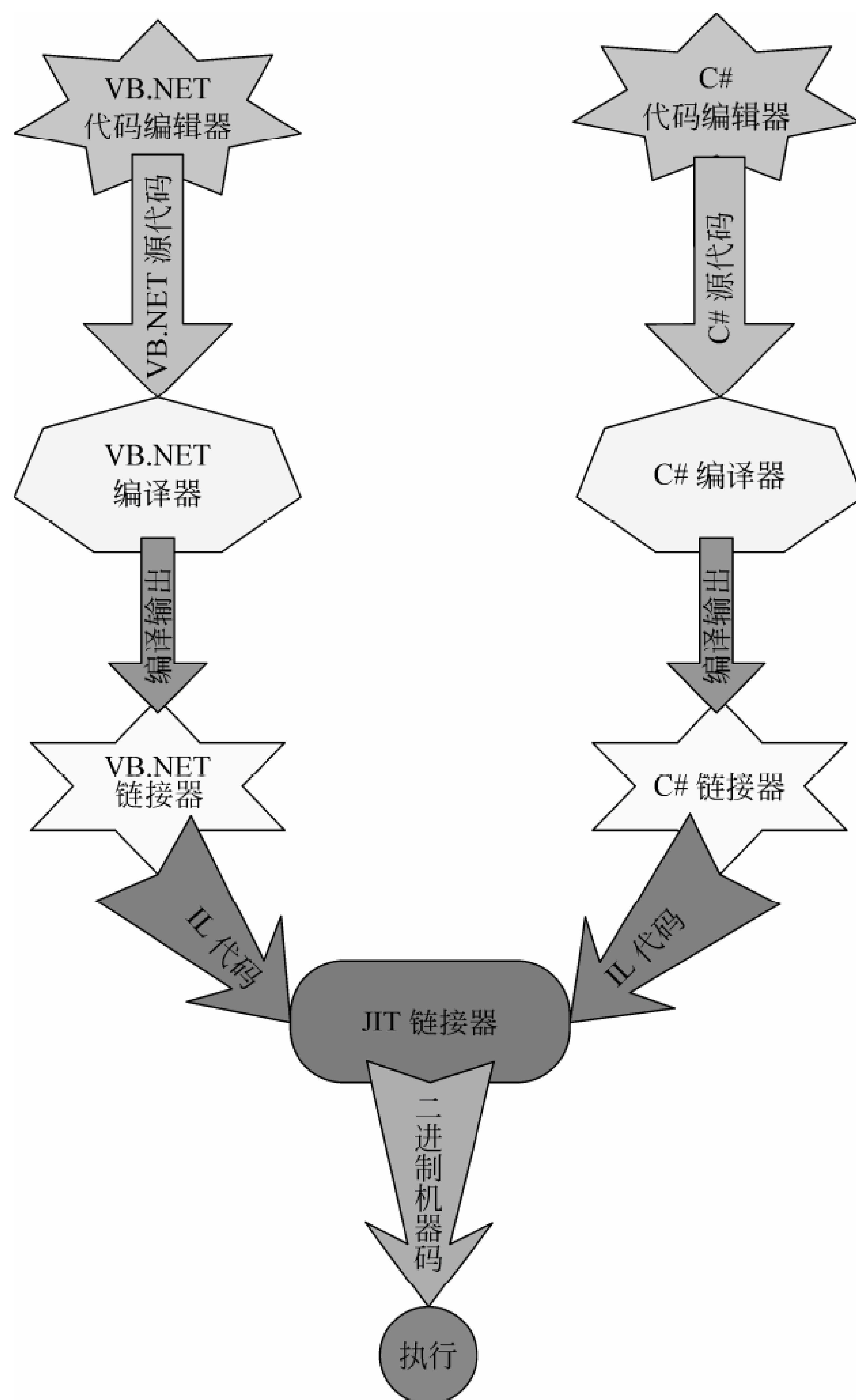


图 1-1　代码从编译到执行全过程

1.2.3　公共语言规范与 C#

.NET 的一个目标之一就是改变多种开发语言各自为政，相互类型上不完全兼容的问题，这就需要实现跨语言编程。多种开发语言要进行相互之间的完全交互，必须定义一种统一的语言规范，不同开发语言编写出来的代码最终被编译成满足该规范的代码，这样任何其他语言编写的程序就可以方便地调用这些代码。

.NET 框架通过公共语言规范（CLS）实现跨语言编程，公共语言规范定义了所有可以在.NET 框架上运行的代码所必须满足的基本接口。CLS 在设计上足够大，可以包括开发人员经常需要的语言构造，同时也足够小，大多数语言都可以支持它。CLS 定义了基本数据类型的数量和占用空间，比如，Int32 表示 4 字节的带符号整数，Int16 则表示 2 字节的带符号整数等。CLS 也定义了面向对象开发语言中的基本元素——类，定义了类的构造函数、成员、可访问性等，CLS 还定义了关于泛型的统一接口。由于 CLS 本身是一个全面的规范，篇幅限制，这里就不再详细介绍。

在微软官方推出的开发语言中，VB.NET 和 C#是最流行的符合 CLS 规范的开发语言，它们也是.NET 应用程序最主要的开发语言，当然也可以用 VC++.NET 来开发.NET 程序。C#是微软专门为.NET 打造的全新的面向对象开发语言，它既简洁，又是类型安全的，开发人员可以通过它来开发各种安全而可靠的应用程序。

C#简单易学，只有 90 个关键字，且使用 C/C++或 Java 里常用大部分语法习惯，C#综合 C++和 Java 的优点，并进行补充，具有更多新特性。从语法上，C#简化了 C++的复杂性，但是功能却比 C++更强大。C#支持空的值类型、枚举、委托、匿名方法和直接内存访问。C#还支持泛型方法和类型，使得类型安全和性能更优。

C#作为一种面向对象的语言，同样支持封装、继承和多态。和 Java 一样，所有的变量和方法，包括应用程序的入口（Main()方法）都封装在一个特定的类中，而在 C++中程序入口是全局的。为了避免多重继承带来的麻烦（在 C++中常遇到），C#中类只能直接从一个父类继承，但它支持接口来达到多重继承的效果。C#还支持结构体，并且结构体是一个值类型，而不是引用类型。

除了基本的面向对象实现之外，C#和 Visual Studio 结合，还提出了以下几种更新的机制来加快软件设计和编码。

- 委托（Delegate）：作者认为可以简单看成是封装好的函数指针，并且支持多个函数指针顺序调用，它是事件机制和函数回调的基础。
- 属性（Property）：是对一个或多个私有字段访问的封装，是进行类数据隐藏和安全访问的基础。
- 属性（Attribute）：该机制提供关于运行时类型的声明性元数据，比如可以通过 FlagsAttribute 属性将一个枚举定义成是按位存储来减少存储空间。
- 内联 XML 文档注释：该机制是一种 XML 格式的代码注释，结合 Visual Studio 2008 的类设计器和代码编辑环境，可以加快类设计，增强代码的可读性。
- C#还提供了指针和不安全代码机制，用在直接内存访问必不可少的情况下，进行一些内存操作。
- C#中没有头文件和源文件概念，对类、接口、委托、方法、字段、属性等定义的位置和顺序没有要求，数量上也没有任何限制。

C#是本书所有例程的开发语言，作者将带领读者由浅入深学习 C#的数据类型、基本语法，学习如何使用 C#进行软件开发。

1.3　.NET 类库

.NET 框架作为 Windows 下现在及未来的主要开发平台，它不仅具有各种高级特性，同时还为开发人员提供了大量免费而且通用的应用程序开发接口。这些应用程序开发接口覆盖了用户界面、数据库操作、XML 操作、文件操作、硬件访问、Web 开发等所有的开发领域。本节介绍相关基础知识。

1.3.1　命名空间和程序集

在.NET 之前，软件版本控制是软件部署人员一件非常头痛的事情，尤其是在模块众多

的复杂软件系统中，部署人员往往要花费大量的时间解决模块版本不一致导致的问题。在.NET 中，程序集作为应用程序的构造块，构成了部署、版本控制、重复使用、激活范围控制和安全权限的基本单元。通过程序集可以很轻松地进行软件系统中各组件的版本控制、使用和部署。

程序集通常以一个动态链接库（DLL）的形式存在，包括公共语言运行库执行的代码，它形成安全边界、类型边界、引用范围边界和版本边界。程序集可以被其他程序集引用，从而实现模块的重用，提高开发效率。

每个.NET 程序集都包含一个或多个命名空间，命名空间是一种用于准确定位数据类型的技术，通过命名空间层层定位，可以明确地找到具体要使用的数据类型（基本类型或自定义类等）。如果将程序中的数据类型看成是文件，那么命名空间就可以看成是文件夹，一个命名空间可以包含具体类型和子命名空间。例如，类型 System.Threading.Timer 和 System.Windows.Forms.Timer 都表示类型 Timer，但前者是线程命名空间 System.Threading 下的 Timer 类，后者是命名空间 System.Windows.Forms 下的 Timer 类。.NET 可以根据命名空间找到两个具有相同名字的类型，从而避免出现混淆和错误。

在实际软件开发过程中，程序集常用来进行模块级别的划分，将相互关联的代码放在同一个程序集中，为其他需要使用的模块或应用程序提供接口以便进一步重用。而在同一个程序集中，命名空间常用来实现更细的划分，同一个接口的不同实现用不同的命名空间来表示。比如，System.Data.SqlClient 和 System.Data.OleDb 是两个不同的命名空间，分别实现 SQL Server 和 OLE 数据库的访问，但是它们都实现同样的数据库操作接口（ADO.NET），这个接口则定义在 System.Data 命名空间下。

注意：程序集和命名空间并没有必然的划分和关系，这里提到的原则只是通常做法，理论上允许不同程序集包含同一个命名空间。合理地使用程序集和命名空间，可以让软件结构和代码层次更加清晰。

1.3.2　垃圾回收器

具有 C++编程经验的读者应该对动态内存所带来的内存泄漏问题记忆犹新，尤其是在复杂的软件系统中，内存的分配和释放变得十分谨慎，开发人员不得不花很多时间去考虑何时该释放内存。在.NET 的公共运行库中提供一种自动内存管理机制用来自动追踪内存对象，并在不需要的时候自动释放内存，这就是垃圾回收机制。

有了垃圾回收机制，开发人员不再需要关心对象在什么时候需要释放，而只需在需要的时候创建分配一块内存即可（创建一个对象）。在.NET 中，所有的对象都是引用，每一个引用具有一个引用计数器，用来表示该对象（实际占用一片内存）现在被多少个引用所引用。在第一次分配对象的时候引用计数器被置为 1，每当该对象被再次使用时，引用计数器会加 1。而每当对象推出作用域不再有效时，引用计数器减 1。

同时公共运行库本身控制着垃圾回收器的周期性执行，每次回收器执行都会扫描当前被应用程序分配的对象，如果它的引用计数器为 0，则表示它不再被引用，即可以被销毁。这时垃圾回收器会自动释放对象所占用的内存。

通过垃圾回收机制可以很好地防止内存泄漏，同时为了能够更快地释放内存，.NET

还为开发人员提供了显式控制垃圾回收器的接口，可以明确通知垃圾回收器马上进行内存清除，以便马上释放内存。因为垃圾回收器自动释放可能会有一定时间上的延迟，但是在正常情况下这不会影响到软件的运行。

1.3.3　.NET 类库范围

为了让软件开发变得更加轻松和高效，微软在.NET 框架基础之上实现了大量公用的程序集。这些程序集涵盖了软件开发的各个领域，并且提供了非常简单易用的开发接口和文档，通常称为.NET 类库。主要包括以下方面的内容：

（1）基本数据类型，System 命名空间提供了应用程序开发必需的基本数据类型，包括数值类型（如：Int32，Single 等）、字符类型（Char）、字符串类型（String）、枚举类型等。

（2）集合类，System.Collection 命名空间提供了最常用的几种集合类，包括列表、字典、哈希表等。

（3）IO 操作类库，.NET 框架通过 System.IO 命名空间提供了大量的 IO 操作相关类库，封装了串口的访问、文件和目录的访问等操作。

（4）Windows 用户界面类库，System.Windows 命名空间下包含了所有的 Windows 窗体程序所需要的类，它们主要是 Windows 窗体（Form）和 Windows 窗体控件（Contorls），还包括界面操作相关的类。

（5）ADO.NET 类库，ADO.NET 是.NET 下新型的数据库访问模式，它由 System.Data 命名空间提供支持，封装了多种数据库访问方式。这是本书的重点内容。

（6）LINQ 类库，System.Linq 命名空间提供 LINQ 技术的相关操作类型，这也是本书的内容之一。

（7）多线程相关类库，.NET 中通过 System.Threading 命名空间提供多线程开发类库，包括线程创建和使用、线程同步机制、定时器等多种技术的封装。

（8）Internet 开发相关类库，在.NET 中通过 System.Net 命名空间提供了 TCP/IP 网络开发相关类库，通过这些类库可以进行常见的 TCP/IP 服务器和客户端的开发，完成数据的收发。此外，还可以通过封装的 FTP 协议方便地进行 FTP 服务器和客户端的开发。

（9）ASP.NET Web 控件类库，.NET 框架通过 System.Web 命名空间提供了丰富的 Web 页面开发相关的类库，包括邮件服务、页面缓存、Web 页面控件等多种类库，通过这些类库可以方便地进行网页开发。

除了上面提到的这些主要的类库外，.NET 类库还提供了本地化、序列化、安全性、用户认证等更多类库，由于篇幅所限，本书就不做详细介绍了。

1.4　小　　结

.NET 框架作为微软推出，在 Windows 下现在以及将来最流行的开发平台之一，它将 Windows 下的应用程序开发接口统一到一起。丰富而通用的.NET 类库隐藏了不同操作系统、不同通信技术、不同硬件接口等对开发带来变化的因素，让应用程序开发变得更加简

单而高效。

另外，微软还推出了公共语言规范，以及微软中间代码等概念，使得.NET 应用程序也可以实现跨语言和跨平台。C#和 VB.NET 就是两大主流的.NET 开发语言，尤其是 C#更是收到广泛的关注和使用。在进一步学习 C#开发语言，以及 ADO.NET、LINQ 等高级开发技术之前，本章从.NET 框架的历史开始，简单介绍了.NET 框架的基础知识和重要特性。通过本章的学习，读者应该掌握以下知识点：

- 微软为什么要推出.NET？
- .NET 经历了哪些版本？各有什么特点？
- 为什么要提出公共语言规范？它规定了哪些内容？
- 中间语言在.NET 框架中处于什么角色？
- .NET 应用程序从编译到运行经历什么样的过程？
- .NET 如何实现命名空间分离？程序集有什么好处？
- .NET 框架中垃圾回收机制如何工作？
- .NET 类库包括哪些方面的接口？

第 2 章　C#基本语法

随着软件开发技术的不断发展，编程语言也不断朝着高级和应用层发展。从最底层的汇编语言到结构化的 C 语言，再到面向对象编程语言 C++和 Java，开发语言越来越高效和快捷，语言本身的功能也越来越强大。本书将为读者介绍一种语法简洁而功能强大的面向对象编程语言——C#。

2.1　开发第一个 C#程序

C#是一种简单易学、功能强大的面向对象语言，主要用于开发在.NET 环境下运行的各种应用程序，它的程序结构简单易懂。本节将介绍 C#程序的基本结构。

2.1.1　创建控制台应用程序

Visual Studio 2010 是微软开发的软件开发集成环境，通过它能够进行包括 C#、VB.NET、VC++.NET 等多种语言在内的多种应用程序开发。Visual Studio 系列 IDE 具有非常悠久的历史，也经历了多个版本的改进，以良好的用户界面著称，是 Windows 下进行软件开发必不可少的利器。由于篇幅关系，本书并不介绍 Visual Studio 开发环境的安装及介绍，而是假设读者已经了解并有一定的 Visual Studio 使用经验，读者可以从 MSDN 或 Visual studio 官方网站了解更多详细信息。

通常都是通过 Visual Studio 2010 创建一个 C#应用程序，只需要如下几个简单步骤即可。

（1）通过开始菜单打开 Visual Studio 2010 开发环境，进入初始界面。

（2）通过菜单“新建”|“项目”打开创建项目对话框，如图 2-1 所示。该对话框的 6 个主要部分用 1～6 标识起来。Visual Studio 2010 支持开发多种.NET 版本的程序，在标识 1 处可以选择.NET 版本，包括.NET 2.0、.NET 3.0、.NET 3.5、.NET 4.0 这 4 个版本可选。在标识 2 处选择要创建的项目的开发语言，可见 Visual Studio 2010 可以开发多种语言的应用程序。在标识 3 处输入应用程序名称，标识 4 处输入应用程序的存放的路径，也可以通过标识 6 处的“浏览”按钮选择已有目录。标识 5 表示是否需要创建解决方案目录。

（3）在图 2-1 中填入相关参数后，单击“确定”按钮，Visual Studio 2010 会根据所选模板自动创建应用程序的主要代码。在本例中，选择语言 C#，.NET 版本为 4.0，模板为最简单的“控制台应用程序”，创建解决方案目录。

通过上面 3 个步骤，就成功创建了一个最基本的基于控制台的 C#应用程序，并且通过 Visual Studio 2010 的“调试”|“开始执行”菜单命令可以运行程序。当然这个时候程序还

没有任何实际功能（输出结果），但是程序的结构却已经成型。

2.1.2　分析 C#程序结构

在通过 Visual Studio 2010 创建了应用程序之后，它会自动创建好应用程序的基本结构，并产生对应的文件和目录。一个 C#应用程序包含两个基本概念：解决方案（Solution）和工程（Project）。解决方案可以理解成是一个解决问题的一整套方案，它包含多个相互关联的模块。这些模块则可以看成是一个工程，只有在所有的子模块解决后，整个问题才会得以解决。在最简单的情况下，一个解决方案只包含一个工程。

通常，一个 C#应用程序用一个解决方案来表示，而它可能包含多一个工程。工程和解决方案之间这种关系，也被表现在应用程序的目录结构上。

- 解决方案目录：在该目录下通常包含一个解决方案文件（*.sln），该文件是一个 XML 文件，记录了当前解决方案中所包含的所有工程名称、相对路径等。在创建应用程序时可以指定是否创建独立的解决方案目录。
- 工程目录：该目录下通常包含一个工程文件（*.csproj），该文件也是 XML 文件，记录一个工程的名称、编译调试等配置信息。该目录通常还包含一个 bin 和 obj 目录，分别是默认的目标文件和中间文件输出路径；还包含一个 Properties 目录，用来记录工程的版本信息等属性。

一个 C#应用程序，只能有一个主工程，该工程包含了应用程序的入口 Main()函数（更多细节在 2.4.2 节介绍），最简单的 C#应用程序代码结构如图 2-2 所示。

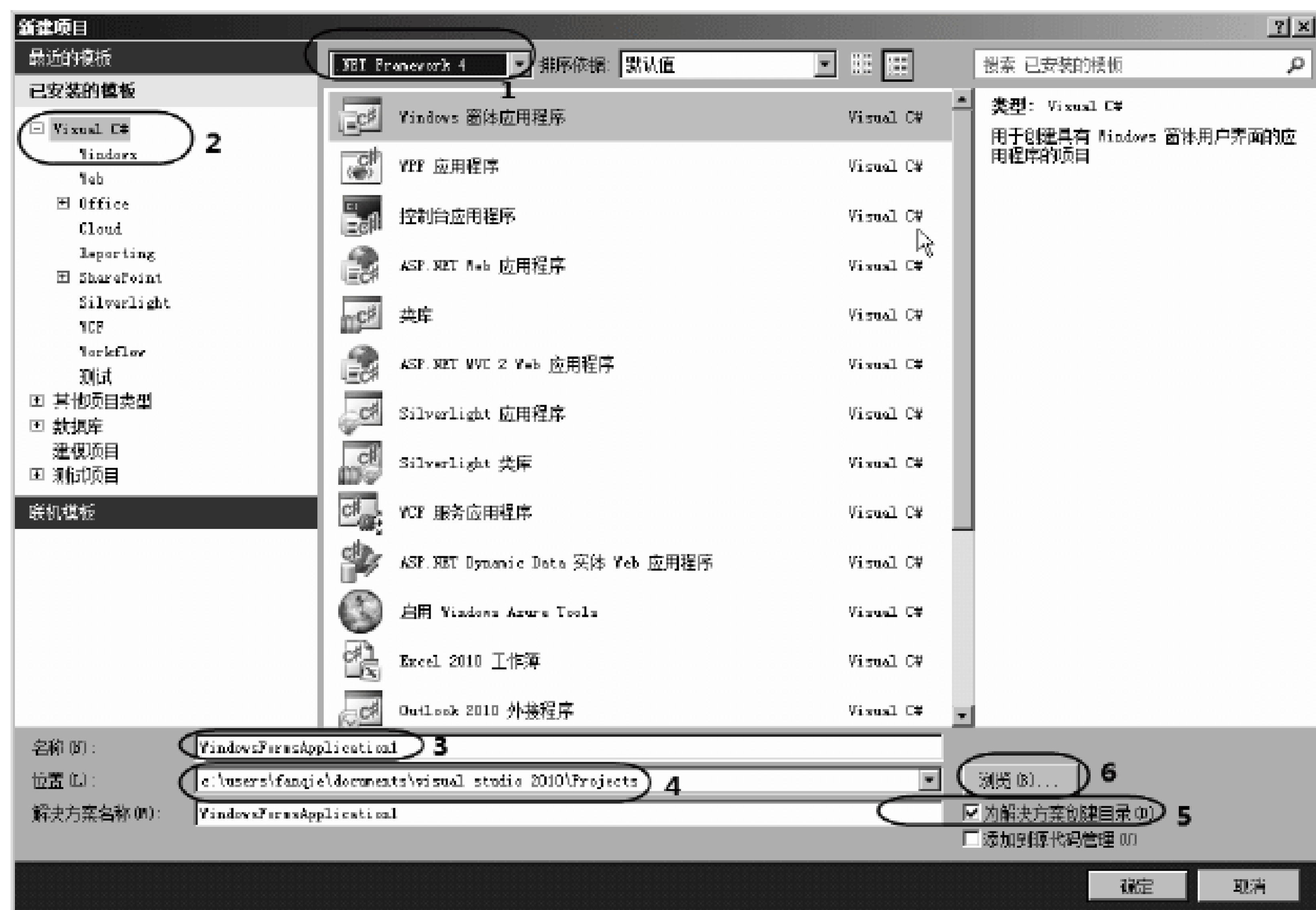


图 2-1　新建项目对话框



图 2-2　应用程序代码结构

其中，“引用”下面是该程序正常运行需要用到的其他类库，这些类库可以是.NET 类库，也可以是自定义类库。图 2-2 中列出的是 Visual Studio 2010 自动添加的所有 C#应用程序都需要用到的类库。Properties 下面的 AssemblyInfo.cs 是版本信息代码文件，在这里可以配置工程的版本、公司、版权等信息，如示例代码 2-1 所示。

示例代码 2-1

```
using System.Reflection;
```

```
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

//有关程序集的常规信息通过下列属性集
//控制。更改这些属性值可修改
//与程序集关联的信息。
[assembly: AssemblyTitle("FirstCSharpProg")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("LuckyMouse")]
[assembly: AssemblyProduct("FirstCSharpProg")]
[assembly: AssemblyCopyright("Copyright © LuckyMouse 2008")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//将 ComVisible 设置为 false 使此程序集中的类型
//对 COM 组件不可见。如果需要从 COM 访问此程序集中的类型，
//则将该类型上的 ComVisible 属性设置为 true。
[assembly: ComVisible(false)]

//如果此项目向 COM 公开，则下列 GUID 用于类型库的 ID
[assembly: Guid("13ed8813-4860-4a2c-89a8-985cb63fd0e7")]

//程序集的版本信息由下面四个值组成：
//主版本
//版本
//内部版本号
//修订号
//可以指定所有这些值，也可以使用"内部版本号"和"修订号"的默认值
//方法是按如下所示使用"*"
 [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

代码文件 Program.cs 是应用程序真正的主体，它定义了应用程序的入口函数 Main()，开发人员通过在这里添加代码来实现应用程序的功能。如示例代码 2-2 所示。

示例代码 2-2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FirstCSharpProg
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("This is the first C# program....");
            System.Console.WriteLine("这是第一个 C#应用程序....");
        }
    }
}
```

示例代码 2-2 给出了 C#代码结构的典型结构，在代码最前面用 using 语句导入需要使

用的命名空间，然后用 namespace 语句定义当前代码所在的命名空间，通过 class、enum 等定义数据类型。这里的 Main()函数是应用程序的入口，最后一个代码文件包含该函数。

注意：这里的 Program.cs 只是 Visual Studio 2010 自动产生的文件名，实际上的项目中具有很多文件名，而且完全可以修改 Program.cs 为其他文件名，如 Main.cs。

2.1.3　添加 C#代码注释

在 C#中，最常见的代码注释的方法有两种，细心的读者肯定已经从前面的示例代码段中看到了。C#沿用 C++中代码注释方法，包括“/* */”和“//”两种注释方式。

“/* */”方式为代码添加一行或多行注释，注释从“/*”开始，到“*/”结束，如下代码所示。

```
/*只有一行注释*/
int i = 0;
/*
 * 第 1 行注释
 * 第 2 行注释
 */
int j = 0;
```

需要注意的是，为了使注释清楚明了，C#中不支持“/* */”的嵌套注释，如下注释是不合法的。其中，1 和 2 两段都被成功注释，但是第 3 段则由于不支持嵌套注释导致没有得到注释。

```
/* 1.成功被注释
 /* 2.成功被注释*/
 3.由于嵌套，没有被注释到。
*/
```

在 C#中，还可以用“//”添加单行注释，在“//”之后的同一行中所有文本都认为是注释。如下代码所示。

```
int i = 0; //单行注释
int j = 0; //另外一行注释
```

技巧：代码注释讲究工整合理，不要滥用，也不能不用。为了避免混乱，尽量不要嵌套使用“/* */”和“//”进行注释。尽可能地使用“//”进行注释。

2.2　变量和数据类型

2.1 节简单介绍了 C#应用程序的代码结构和目录结构，并介绍了最基本的应用程序类型——控制台应用程序。从本节开始，将带领读者进入 C#的世界，学习更多 C#的语法和开发技巧。

2.2.1　定义变量

作者认为，软件开发是一个将现实中的各种具体或抽象的数据数字化的过程，软件的目的是将所有数据都以二进制数据的形式表示并进行数学运算，从而实现软件特定的功能。这就需要开发语言本身能够表示多种数据，以及同一类数据的多个实例（个体）。

和其他开发语言一样，C#也采用变量来定义和保存某个具体的数据，变量具有 3 个基本要素：变量名、数据类型和变量值。一个变量只有在声明之后才能被使用，在 C#中使用如下格式定义一个变量：

```
DataType varName [= initValue];
```

其中，DataType 表示变量的具体类型，表示该变量的数据应该以哪种方式进行计算和处理。varName 表示变量的名称，在同一个作用域内，它用来唯一标识一个变量，就如一个人的姓名。initValue 可选，如果指定了初值，那么这个变量在定义之后就为该值，否则变量的值不可预期，不能直接使用。

如示例代码 2-3 中，定义了 3 个变量，intVal1 和 intVal2 都是 int 类型的整数变量，但是 intVal1 没有初值，所以在定义之后赋值之前，它的值是不确定的，只有在赋值之后才能使用。而 intVal2 则在定义时指定了初值为 1，那么此时 intVal2 的值为 1，而且可以直接使用。而 txt 是 string 类型的字符串变量，并指定初始值为“Hello World”。

示例代码 2-3

```
int intVal1;                    //定义 int 类型变量 intVal1
int intVal2 = 1;                //定义 int 类型变量 intVal2 并初始化为 1
string txt = "Hello World";     //定义字符串变量 txt，并初始化为"Hello World"
```

在 C#中，还可以在同一个声明语句中定义多个类型相同的变量，并且可以分别为它们指定初值，多个变量采用逗号（,）分隔。如示例代码 2-4 所示，同时定义 2 个 int 类型变量 intVal1 和 intVal2，前者没有初值，后者初值为 1。第二行代码同时定义 3 个 string 类型变量 txt1、txt2、txt3，并分别制定值。

示例代码 2-4

```
int intVal1, intVal2 = 1;
string txt1 = "Hello World", txt2, txt3 = "I have a dream. ";
```

在 C#中，变量名是区分大小写的，而且需要满足以下规则：

- ❑ 变量名必须以字母，下划线或@符号开头。
- ❑ 变量名只能由字母、数字和下划线组成，而不能包含空格、标点符号、运算符等其他符号。
- ❑ 变量名不能与 C#中的关键字名称相同，关键字是指 C#语法中已经使用的词语，如 if、else 等。
- ❑ 变量名不能与 C#中的库函数名称相同。

技巧： 作者建议使用具有实际意义的变量名，避免使用 i、j、k 之类的名称，这样可以大大增加代码的可读性和可维护性。另外，变量在需要使用的时候再去定义，避免定义无用的变量。

2.2.2　使用数值类型

在软件中，所有数据都是以二进制的形式保存在内存中，内存中相同的数据可以被解释为不同的意思，数据类型就是用来定义内存中数据的具体解析格式。比如：内存的一个字节值为 0xFF，如果它是有符号整数，值为–1，如果它是无符号整数，值为 255。

在 C#中，数据类型定义了数据的具体格式，在代码执行过程中根据数据类型为变量分配空间和计算变量的值。C#中包含最基本的数据类型：数值、字符、字符串等，用来表示日常生活中常见的数据。同时提供将这些基本类型组合起来定义自定义类型的机制，从而使得 C#可以表示生活中的所有具体或抽象的数据。

C#中，数值类型用来表示数字——整数和小数。其中，根据数值的符号和范围不同，C#包含表 2-1 所示的整数类型和表 2-2 所示的小数类型，由此可见，C#可以表示生活中能遇到的所有数值。

表 2-1　C#中的整数

数据类型	字节数	取 值 范 围	说　　明
byte	1字节	0～255	8位无符号整数
sbyte	1字节	–128～127	8位带符号整数
ushort	2字节	0～65 535	16位无符号整数
short	2字节	–32 768～32 767	16位带符号整数
uint	4字节	0～4 294 967 295	32位无符号整数
int	4字节	–2 147 483 648～2 147 483 647	32位带符号整数
ulong	8字节	0～18 446 744 073 709 551 615	64位无符号整数
long	8字节	–9 223 372 036 854 775 808～9 223 372 036 854 775 807	64位带符号整数

表 2-2　C#中的小数

数据类型	字节数	取 值 范 围	精　　度	说　　明
float	4字节	±1.5*10e-45～±3.4*10e38	7位小数	32位小数
double	8字节	±5.0*10e-324～±1.7*10e308	15到16位小数	64位小数，范围大，精度较低
decimal	16字节	±1.0*10e-28～±7.9*10e28	28到29位小数	128位小数，范围不大，但是精度极高，适合财务和货币运算

示例代码 2-5 结合变量的定义演示数值类型变量的使用，并通过程序打印出各个数值类型的取值范围，其实这里也用到了字符串类型，将在 2.2.3 节介绍。

示例代码 2-5

```
static void PrintDataRange( )
{
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "byte",
```

```
    byte.MinValue, byte.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "sbyte",
    sbyte.MinValue, sbyte.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "ushort",
    ushort.MinValue, ushort.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "short",
    short.MinValue, short.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "uint",
    uint.MinValue, uint.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "int",
    int.MinValue, int.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "ulong",
    ulong.MinValue, ulong.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "long",
    long.MinValue, long.MaxValue);

    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "float",
    float.MinValue, float.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "double",
    double.MinValue, double.MaxValue);
    System.Console.WriteLine("\"{0}\" 的取值范围为: {1} 到 {2}", "decimal",
    decimal.MinValue, decimal.MaxValue);
}
```

示例代码 2-5 的输出为：

```
"byte" 的取值范围为: 0 到 255
"sbyte" 的取值范围为: -128 到 127
"ushort" 的取值范围为: 0 到 65535
"short" 的取值范围为: -32768 到 32767
"uint" 的取值范围为: 0 到 4294967295
"int" 的取值范围为: -2147483648 到 2147483647
"ulong" 的取值范围为: 0 到 18446744073709551615
"long" 的取值范围为: -9223372036854775808 到 9223372036854775807
"float" 的取值范围为: -3.402823E+38 到 3.402823E+38
"double" 的取值范围为: -1.79769313486232E+308 到 1.79769313486232E+308
"decimal" 的取值范围为: -79228162514264337593543950335 到
79228162514264337593543950335
```

2.2.3　使用字符串类型

除数值外，日常生活中另外一个最基本的数据是文本，比如说的话、读的书、写的作品等都是用文本表示。在 C#中，使用字符串类型表示文本，字符串又可以看成是多个字符的依次连接。关键字 string 表示字符串类型，字符串常量用英文双引号（""）括起来。字符用关键字 char 表示，字符常量用英文单引号（''）括起来。

示例代码 2-6 定义字符变量 ch1 和 ch2，字符串变量 str1 和 str2，可以看出，C#中 char 和 string 都表示的是 Unicode 码，可以直接表示英文和中文，也可以表示任何国家的文字。由于是 Unicode，所以每个 char 占用内存中的两个字节，大小为 0x0000-0xFFFF。

示例代码 2-6

```
char ch1 = 'A';
```

```
char ch2 = '你';
string str1 = "How are you. ";
string str2 = "你好。";
```

由于双引号是字符串在代码中的起始和结束标志，所以直接将双引号添加到字符串会产生混乱，如“A”“B”就是错误的写法。那么，如何在字符串中表示双引号呢？答案是转义字符。

在 C#中，转义字符“\”之后的字符失去自身的含义，而是和“\”一起表示特殊含义。如“\n”表示换行，“\t”表示制表符等。表 2-3 列出了所有转义字符“\”之后只能出现的字符及其含义。

表 2-3　C#转义字符后的字符

字　符	含　义
\\	表示斜杠 \ 本身，如'\\'表示字符斜杠
\'	表示单引号 ' 本身，只能出现在字符中，如'\' '表示字符单引号
\"	表示双引号 " 本身
\0	表示后面跟着的是八进制数，该数表示一个字符
\a	表示蜂鸣器运算符
\b	表示回退符号
\f	表示进纸符号
\n	表示换行符
\r	表示回车符
\t	表示横向制表符
\u	表示一个数值表示的Unicode字符序列
\x	表示后面跟着十六进制数，该数表示一个字符
\v	表示垂直制表符

示例代码 2-7 演示转义字符的具体使用实例。第 1 个字符串由于\b 出现，导致第 1 个字符 C 被回退键删除。第 2 个字符串中\x64 是字符 d 的十六进制值，所以输出为“Abd”。第 3 个字符串中\t、\'、\"都被转义。第 4 个字符串中\n 表示换行，所以最终输出为两行。

示例代码 2-7

```
static void StringFunc()
{
    System.Console.WriteLine("ABC\bC");
    System.Console.WriteLine("AB\x64");
    System.Console.WriteLine("AB\tC\'D\"EF");
    System.Console.WriteLine("AB\nCDEF");
}
```

示例代码 2-7 的输出如下：

```
ABC
ABd
AB      C'D"EF
AB
CDEF
```

2.2.4　使用枚举和布尔类型

C#中还提供一种数据类型叫布尔类型，用关键字 bool 表示，它有且仅有两个值 true 和 false，用来表示是/否、对/错等关系。如果两个 bool 变量都为 true 或都为 false，那么这两个变量认为是相等，否则认为不等。此外，布尔类型可以进行逻辑运算，将在 2.3.4 节介绍。

bool 类型只有两个值，很多时候不够用。因此，C#支持一种简单的自定义类型——枚举类型，枚举就是将一组相关的可选值定义成一个数据类型，该数据类型的值不超过定义的范围。通过 enum 关键字定义枚举，其格式如下：

```
enum typeName
{
    Name1 [= 整数值],
    Name2 [= 整数值],
    ...
    NameN [= 整数值][,]
};
```

其中：

- typeName：表示枚举类型的名称，往往是一个具有实际意义的字符串，如 States、Modes 等。
- Name1～NameN：表示该枚举类型的 N 个可选项的名称，每个可选项可以指定具体的整数值，如果不指定，默认为前一个可选项的整数值加 1。第 1 个可选项（Name1）的值默认为 0。
- 可选项之间用“,”分隔，最后一个可选项之后的“,”分隔符可以不要。

枚举类型的可选项可以看成是一些相关的整数常量。枚举类型变量则可以简单看成是取值范围有限的整数，所以枚举和整数可以相互转换，可以对枚举进行加、减、比较操作。如果枚举取值在可选范围内，则打印为枚举可选项的名称，否则直接打印整数值。示例代码 2-8 演示如何在 C#中定义和使用枚举。

示例代码 2-8

```
class Program
{
   public enum Weeks
   {
      Monday = 1,   //周一，特殊指定，值为 1
      Tuesday,      //周二，默认递增，值为 2
      Wednesday,    //周三，默认递增，值为 3
      Thursday,     //周四，默认递增，值为 4
      Friday,       //周五，默认递增，值为 5
      Saturday,     //周六，默认递增，值为 6
      Sunday = 0,   //周日，特殊指定，值为 0
   }

   static void Main(string[] args)
   {
```

```
        Weeks day = Weeks.Monday; //day=1
        System.Console.WriteLine("day = {0}", day);
                                        //在枚举范围内，打印枚举名字符串
        day++; //枚举添加实际就是整数相加，此时 day=2
        System.Console.WriteLine("1: day = {0}", day);
        day = Weeks.Saturday;           //此时 day=6
        day++; //此时 day=7
        System.Console.WriteLine("2: day = {0}", day);
                                        //超出枚举指定范围，直接打印整数
        System.Console.WriteLine("3: day = Moday : {0}", day == Weeks.
        Monday);
    }
}
```

在示例代码 2-8 中，定义枚举类型 Weeks 用来表示一个星期的 7 天，星期一到星期六分别用数字 1～6 表示，而星期天则用 0 表示。然后定义一个 Weeks 类型的变量 day，初始化为 Monday。然后对 day 进行加、等于操作。该代码的输出如下：

```
1: day = Monday
2: day = Tuesday
3: day = 7
4: day = Moday : False
```

提示：可以通过强制类型转换将一个整数转换成枚举类型，比如 Weeks day=(Weeks)6;将整数 6 直接转换成枚举值 Sunday。枚举类型向整数的转换是自动的，如 int val = day;执行之后 val 的值将变成 Sunday 的整数值（即 0）。

2.2.5　定义和使用结构体类型

前面小节中介绍的所有数据类型都属于 C#提供的最基本的数据类型，然而仅仅通过这些数据类型还远远不能解决复杂问题。通过 C#可以定义结构体和类，开发人员通过这两种机制可以定义自己需要的数据类型。由于结构体和类有很多相似之处，而且类是 C#语言的一个重点，所以本节只是简单介绍结构体，关于结构体和类的更多内容请阅读本书第 3 章。

结构体可以看成是一组具有某种关系的数据组合，它们作为一个整体描述了某种事物或状态。如猫具有名称、颜色、体重 3 个信息，这 3 个信息可以组合为结构体表示猫的特性。C#中，通过关键字 struct 定义结构体，结构体的具体内容用大括号“{}”括起来。代码如下：

```
struct StructName
{
    member list
}
```

其中，StructName 表示结构体的名称，通常是一个具有实际意义的名词，该名称就如同 int、string 一样用来表示定义的新类型。通常，结构体只是用来保存数据，所以它可以包含各种类型的字段，也可以包含读写这些字段的属性。如示例代码 2-9 中定义结构体 Cat，它具有一个私有（private）的 string 类型字段_Name，表示猫的名称，同时包含一个公开（public）的属性 Name，用来获取或设置_Name。另外，它还包括两个公开（public）的字

段 Weight 和 Colour，分别表示体重和颜色。

示例代码 2-9

```
struct Cat
{
   private string _Name;
   public string Name
   {
      get
      {
         return  Name;
      }
      set
      {
          Name = value;
      }
   }
   public int Weight;
   public string Colour;
}
```

定义了结构体 Cat 之后，就产生了 Cat 这样一个类型，就可以像使用 int 一样使用 Cat，可以定义 Cat 类型的变量，可以通过点运算符“.”访问 Cat 类型的公开成员。在结构体外部不能访问结构体的私有成员，只能访问公开成员，关于私有和公开可访问性的细节可参看第 3 章。示例代码 2-10 演示结构体 Cat 的使用。

示例代码 2-10

```
static void Main(string[] args)
{
   Cat ACat;                    //定义 Cat 类型变量
   ACat = new Cat( );           //新建 Cat 对象，并设置为 ACat 的值
   ACat.Name = "Tom";           //通过属性 Name 设置 ACat 的名称
   ACat.Weight = 5;             //通过字段 Weight 设置 ACat 的体重
   ACat.Colour = "Red";         //通过字段 Colour 设置 ACat 的颜色

   //获取 ACat 的值，并打印到控制台
   System.Console.WriteLine("This is CAT {0}, Weight is {1}, Colour is {2}",
       ACat.Name, ACat.Weight, ACat.Colour);
}
```

如示例代码 2-10 所示，在定义 Cat 类型的变量 ACat 之后，在使用这个变量之前还需要进行赋值。在 C#中通过关键字 new 来创建新的结构体对象，执行 new 操作根据 Cat 类型的定义从内存中分配一个内存区域，用来保存 Cat 结构体的值。示例代码 2-10 的输出如下所示：

```
This is CAT Tom, Weight is 5, Colour is Red
```

除了字段和属性之外，结构体同样可以包含构造函数、方法、事件、索引器、嵌套类型等，但是如果结构体同时具有多个上面的成员，那么该结构体可以考虑用类来实现了。结构体在 C#中是值类型，所以更多是用来传递和保存数据，并不需要行为，而且结构体的字段通常是定义为公开（public）的。

2.2.6　定义和使用数组

在 C#中，数组表示一组相关数据的集合，集合中所有元素具有相同数据类型，而且元素个数通常是在创建数组时就明确的。比如：某公司有 100 个员工，那么 100 个员工的姓名就是一个集合，在 C#中，可以定义一个包含 100 个 string 值的数组来表示，之所以用 string 数组是因为姓名是文本类型。

1．一维数组和多维数组

在 C#中，不仅支持一维数组，还可以定义多维数组，数组的声明格式为：

```
DataType[ , ] ArrayName;
```

其中：

- DataType：表示数组中元素的数据类型，可以是基本类型、自定义类型、数组类型等任何类型。
- ArrayName：表示数组的名称。
- “[]”表示该定义为数组，用“,”表示维数。如“[ ]”表示一维数组，“[,]”表示二维数组，依次类推。

数组变量在声明之后只是确定了类型和维数，在使用前还需要指定其元素个数，可以通过“new”关键字为其指定元素个数，并通过“{}”提供初始值对数组进行初始化。

示例代码 2-11 中，第一行声明一维数组变量 ary1，并通过“new”为其分配 10 个元素的长度。后两行都声明一维数组变量 ary2，并通过“{}”初始化 ary2 的数据为{1，2，3}，ary2 的长度是初始化元素的个数——3。

示例代码 2-11

```
int[] ary1 = new int[10];
int[] ary2 = new int[]{1,2,3};
int[] ary2 = {1,2,3,4,5};
```

示例代码 2-12 声明两个二维数组 d2Ary1 和 d2Ary2，通过“new”指定 d2Ary1 为 3 行 2 列，通过“{}”将 d2Ary2 初始化为 2 行 3 列，第一行值为{1，2，3}，第二行值为{4，5，6}。

示例代码 2-12

```
int[ , ] d2Ary1 = new int[3,2];
int[ , ] d2Ary2 = new int[ , ]{ {1,2,3}, {4,5,6} };
```

多维数组的维数在理论上是任意的，只是通常只采用二维数组，适当时候使用三维数组会有带来很多方便，但是不要使用三维以上的数组。

2．交错数组

C#还支持一种更加灵活的数组——交错数组，交错数组是指元素本身就是数组的数组。交错数组声明格式为：

```
DataType[ ][ ] ArrayName;
```

其中，第一个“[]”定义交错数组本身的维数。后一个“[]”定义交错数组元素的维数。示例代码 2-13 声明一维交错数组变量 jadAry1，并分配长度为 10 个元素。初始化 jadAry1 的第 0 个元素为长度为 3 的一维数组，jadAry1 的第 2 个元素被初始化为数据是{1，2，3，4，5}的一维数组。

示例代码 2-13

```
Int[ ][ ] jadAry1 = new int[10][];
jadAry1[0] = new int[3];
jadAry1[2] = new int[]{1,2,3,4,5};
```

交错数组也同样可以通过“{}”来初始化，示例代码 2-14 声明一维交错数组 jadAry2，并通过“{}”初始化它的两个元素依次为一维数组{1，2，3，4，5，6}和一维数组{1，3，5，7，8}。

示例代码 2-14

```
int[ ][ ] jadAry2 =
{
   new int[] {1,2,3,4,5,6},
   new int[] {1,3,5,7,8},
};
```

3. 访问数组元素

在 C#中，通过“[]”操作符访问数组中的元素，格式为：ArrayName[,]，在“[]”中给出从 0 开始的索引（即 0 表示第 1 个元素，1 表示第 2 个元素，依次类推）。示例代码 2-15 定义一维数组 ary1，并将第 2 个元素（索引为 1）赋值为 10。再定义一个二维数组 ary2，并将第 2 行第 2 列的元素赋值为 11。

示例代码 2-15

```
int[] ary1 = {2,6,8};
ary1[1] = 10;
int[,] ary2 = { {2,6,8}, {3,7,10} };
ary2[1, 1] = 11;
```

另外，在 C#中还可以通过 foreach 关键字从第 0 个元素开始依次遍历数组中的所有元素。

注意： foreach 操作对多维数组元素的遍历是先按行再按列遍历的。

示例代码 2-16

```
static void Main(string[] args)
{
   int[] ary1 = {1,2,3};
   ary1[1] = 5;
   System.Console.WriteLine("Ary1: ");
   foreach (int val in ary1)
   {
      System.Console.Write("{0},", val);
   }
```

```
    System.Console.WriteLine( );

    int[,] ary2 = { {1,2,3}, {4,5,6} };
    ary2[1, 1] = 10;
    System.Console.WriteLine("Ary2: ");
    foreach (int val2 in ary2)
    {
        System.Console.Write("{0},", val2);
    }
    System.Console.WriteLine( );
}
```

示例代码 2-16 中，首先定义并初始化两个数组 ary1 和 ary2，然后通过“[]”修改其元素值，最后通过 foreach 遍历并打印出这些元素的值。程序的输出如下，对二位数组 ary2 的遍历是先行后列的顺序执行。对一维数组 ary1 的遍历是从第 0 个元素到最后一个元素执行。

```
Ary1: 1,5,3,
Ary2: 1,2,3,4,10,6,
```

2.2.7　定义常量

除变量之外，C#还有另外一种存储数据的方式——常量。顾名思义，常量的值是固定不变的，且不可修改的。通过两种方式来使用常量：

（1）编码时直接输入常量的值，在示例代码 2-17 中整数“10”，字符串“Hello World.”都是常量。

示例代码 2-17

```
int intVal2 = 10;            //定义 int 类型变量 intVal2 并初始化为 10
string hello = "Hello World";
                             //定义字符串变量 hello,并初始化为"Hello World. "
```

（2）通过关键字 const 指定一个常量，它的使用格式如下：

```
const DataType VarName = Value;
```

其中，const 是关键字，表示此处定义一个常量。DataType 是常量的数据类型，只能是基本数据类型，不能是自定义类型。VarName 是该常量的名称，命名规则和变量名一样，Value 是该常量的值，一经设置，该常量在整个生命周期都只能是这个值，不能改变。

常量的定义必须指定其初值，而且只能在声明时被赋值一次。示例代码 2-18 定义圆周率常量 Pi，然后在计算面积时使用。

示例代码 2-18

```
const float Pi = 3.1415926;     //定义圆周率常量
float area = Pi * 2 * 2;        //计算半径为 2 的圆面积
```

技巧：对于一些固定的数据，尤其是在多个地方重复使用的固定数据，作者建议使用常量来表示它们。一方面常量名称具有特定意义，可以增加代码的可读性和可维护性。另一方面，当该值需要发生改变时，可以只修改常量的初始值就完成所有代码的修改，快速不遗漏。

2.3 运 算 符

运算符也是编程语言中最基本的元素之一，编程语言用运算符来表示各种算术和逻辑运算，包括加、减、乘、除，也包括或、并等，本节将介绍 C#所支持的大部分运算符。

2.3.1 运算符分类

在 C#中，运算符是指定特定运算的符号，它接受一个或多个操作数，通过对这些操作数执行计算得到返回结果。运算符和它的操作数一起构成表达式，运算结果就是表达式的值。运算符的操作数可以是常量和变量，也可以是表达式本身。

按照运算符操作数的个数，可以将 C#运算符分成一元运算符、二元运算符和三元运算符。一元运算符对一个操作数进行运算，如 ++、--、! 等。二元运算符对两个操作数（左操作数和右操作数）进行运算，如 +、-、* 等。同理，三元运算符对三个操作数进行运算，如最常见的“?:”。

按照运算符的运算功能，可以将 C#运算符分成赋值运算符、算术运算符、比较运算符、逻辑运算符、位运算符等。赋值运算符用于对变量进行赋值。算术运算符对操作数进行加减乘除等算术计算，并返回对应的计算结果。比较运算符对两个操作数进行大小比较，并返回布尔值表示的表达式结果。逻辑运算符用于对两个布尔类型的操作数进行逻辑并、或、与运算，并返回运算结果。位运算符对整数进行按位操作，包括与、并、或、异或、取反等，并返回整数类型的结果。

表 2-4 列出了 C#所支持的主要运算符，以及它们的类型。大部分将在本节详细介绍，还有一些将在后面章节中介绍，如 new、[]、as、is 等。

表 2-4　C#主要运算符

类　别	操 作 数	运 算 符
算术	二元	+、-、*、/、%
算术	一元	++、--
逻辑	二元	&&、||
逻辑	一元	!
位运算	二元	&、|、^、<<、>>
位运算	一元	!、~
比较	二元	==、!=、<、>、<=、>=
赋值	二元	=、+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=
索引	一元	[]
转换	一元	()
条件运算	三元	?:、??
对象创建	一元	new
类型信息	二元	as、is、sizeof、typeof

另外，C#中的运算符都具有优先级和结合方向两个属性。优先级高的运算符优先执行，如表达式“1+3*5”，由于“*”的优先级高于“+”，所以先执行“3*5”，然后再执行“1+”，该表达式的结果为 16。同一个表达式中，相同优先级的两个运算符是根据结合方向执行，左结合则从左到右进行计算，右结合运算方向则正好相反，如表达式“10.0*3.0/5.0”中，由于“*”和“/”优先级相同，且为左结合，则从左到右计算，先计算“10.0*3.0”得 30.0，然后再计算“30.0/5.0”，表达式值为 6.0。

C#中可以通过运算符“()”改变运算优先级。将表达式中需要优先计算的部分括起来，使其能优先执行。如“1+3*5”中要先加后乘，可以写成“(1+3)*5”，这样表达式值就为 20。

2.3.2　用算术运算符进行算术运算

在 C#中，加减乘除是最常见的算术运算符，此外，为了增加算术运算的灵活性，C#还提供了模运算、递增和递减运算。同时还将算术运算和赋值操作相结合，衍生出更加灵活的算术赋值运算符。表 2-5 列出了 C#中算术运算符。

表 2-5　C#中的算术运算符

运算符	功　能	说　明
+、+=	加、加等	执行两个数值的加法运算，后者将结果重新赋值到左操作数
–、–=	减、减等	执行两个数值的减法运算，后者将结果重新赋值到左操作数
、=	乘、乘等	执行两个数值的乘法运算，后者将结果重新赋值到左操作数
/、/=	除、除等	如果是整数相除则返回整数部分（不是四舍五入），小数相除就是普通除法，后者将结果重新赋值到左操作数
%	求余	只对整数进行运算，求一个整数除以另一个整数的余数
++	递增	执行数值的递增，分前缀递增和后缀递增两种
––	递减	执行数值的递减，分前缀递增和后缀递减两种

在算术运算符的使用中，有几个需要特别注意的问题。

1. 数据溢出

在进行算术运算时如果运算结果超出数据类型所能表示的范围，则出现算术溢出，会丢失数据，这个现象常常发生在加法和乘法操作中。

在示例代码 2-19 中，定义两个 uint 类型整数 uval1 和 uval2，将两个变量的和赋值给 uint 类型的变量 usum，由于 uval1 和 uval2 的和为 0x100000010，超出了 uint 的有效范围(0～65535），只保留了低四字节的数据，导致 usum 的值为 0x10。同样的道理，由于 val1 和 val2 的和超出了 int 的范围，sum 就由于数据溢出，变成了负数。

示例代码 2-19

```
uint uval1 = 0xFFFFFFFF;
uint uval2 = 0x11;
uint usum = uval1 + uval2;
System.Console.WriteLine("usum = 0x" + usum.ToString("X"));
```

```
int val1 = int.MaxValue;
int val2 = 10;
int sum = val1 + val2;
System.Console.WriteLine("sum = " + sum.ToString());
long lsum = (long)val1 + val2;
System.Console.WriteLine("lsum = " + lsum.ToString());
```

在实际的编码中，如果不能确定数据是否会溢出，可以先将数据保存到一个范围更大的数据类型中，然后再判断有效性和强制类型转换。如上面代码中将变量 val1 和 val2 的值保存在 long 类型变量 lsum 中，就不会出现数据丢失。注意，首先要强制制定 val1 或 val2 为 long 类型。示例代码 2-19 的输出如下：

```
usum = 0x10
sum = -2147483639
lsum = 2147483657
```

2. 整数除法

在 C#中，整数和小数的除法都通过“/”运算符来完成，但是整数除法只能获得整数部分，且不四舍五入。如“5/15”和“5/6”的值都是 0，因为它们的整数部分为 0，再如“5/2”为 2。小数的除法则返回包括小数的值，如“5.0/10.0”为 0.5。除数和被除数之间任何一个元素为小数类型，都按照小数除法进行运算，如“5.0/10”和“5/10.0”具有同样的结果“0.5”。

3. 递增和递减

递增（++）和递减（– –）运算符都是一元运算符，分别将操作数的值增加 1 和减少 1，根据前缀和后缀不同，表达式的值会有所变化。

- 前缀：操作数的值发生递增或递减，表达式的值是增加或减少后操作数的值。
- 后缀：操作数的值发生递增或递减，表达式的值是增加或减少前操作数的值。

在示例代码 2-20 中，val1 的值为 1，语句“val2=val1++”执行之后，val1 递增变为 2，由于“++”是后缀，val2 的值 val1 递增前的值为 1。语句“val3=++val1”执行之后，val1 递增变为 2，由于“++”是前缀，val3 的值为 val1 递增后的值 2。递减（– –）运算符同样的道理。

示例代码 2-20

```
int val1 = 1;
int val2 = val1 ++;      //val1 = 2, val2 = 1
val1 = 1;
int val3 = ++val1;       //val1 =2, val3 = 2
val1 = 2;
val2 = val1--;           //val1 = 1, val2 = 2
val1 =2;
val3 = --val1;           //val1 = 1, val3 = 1
```

2.3.3 用比较运算符进行比较

在 C#中，比较运算符用于比较两个操作数之间的大小关系，并返回对应的布尔值。比较运算符的两个操作数既可以是常数，也可以是表达式，既可以是数值类型，也可以是字

符和字符串类型，还可以是布尔类型。表 2-6 列出了所支持的比较运算符。

表 2-6　C#比较运算符

运算符	功　能	说　明
==	等于	比较两个操作数是否相等，如果相等为true，否则为false
!=	不等于	比较两个操作数是否不相等，如果不相等为true，否则为false
<	小于	比较第一个操作数是否小于第二个操作数，如果小于为true，否则为false
>	大于	比较第一个操作数是否大于第二个操作数，如果大于为true，否则为false
<=	小于等于	比较第一个操作数是否小于等于第二个操作数，如果小于等于为true，否则为false
>=	大于等于	比较第一个操作数是否大于等于第二个操作数，如果大于等于为true，否则为false

2.3.4　用逻辑运算符进行逻辑运算

在 C#中，逻辑运算符用来对一个或两个布尔值进行逻辑运算，并返回表示结果的布尔值。C#中支持 3 个逻辑运算符：非（!）、与（&&）、或（||），它们的功能如表 2-7 所示。

表 2-7　C#逻辑运算符

运 算 符	功　能	说　明
!	非	一元运算符，对操作数取反。如果操作数为true，结果为false，否则结果为true
&&	与	二元运算符，对两个操作数进行与运算。如果两个操作数都为true，则结果为true，否则结果为false
||	或	二元运算符，对两个操作数进行或运算。如果两个操作数都为false，则结果为false，否则结果为true

示例代码 2-21 演示比较运算符和逻辑运算符的使用，示例中的所有操作数都为常量，是为了使代码更加易读，实际编码中这种写法是很少出现的，操作数通常是变量和表达式。

示例代码 2-21

```
System.Console.WriteLine("8 < 9    :  {0}", 8 < 9);    //true
System.Console.WriteLine("8 > 9    :  {0}", 8 > 9);    //false
System.Console.WriteLine("9 == 9   :  {0}", 9 == 9);   //true
System.Console.WriteLine("10 != 9  :  {0}", 10 != 9);  //true
System.Console.WriteLine("10 >= 9  :  {0}", 10 >= 9);  //true
System.Console.WriteLine("10 > 9   :  {0}", 10 > 9);   //true

System.Console.WriteLine("true && true   :  {0}", true && true);   //true
System.Console.WriteLine("true && false  :  {0}", true && false);  //false
System.Console.WriteLine("false && false :  {0}", false && false); //false
System.Console.WriteLine("true || true   :  {0}", true || true);   //true
System.Console.WriteLine("true || false  :  {0}", true || false);  //true
System.Console.WriteLine("false || false :  {0}", false || false); //false
System.Console.WriteLine("! true         :  {0}", ! true);         //false
System.Console.WriteLine("! false        :  {0}", ! false);        //true
```

代码 2-21 的输出如下所示，从结果中，可以更加明确地理解表 1-6 和表 1-7 中的描述。

```
8 < 9    :  True
8 > 9    :  False
9 == 9   :  True
```

```
10 != 9      :  True
10 >= 9      :  True
10 > 9       :  True
true && true    :  True
true && false   :  False
false && false  :  False
true || true    :  True
true || false   :  True
false || false  :  False
! true          :  False
! false         :  True
```

2.3.5　用位运算符进行位操作

在 C#中，通过位运算符实现二进制数据的按位操作，位运算符接受一个或两个整数类型的操作数，返回位运算后的整数，表 2-8 列出了 C#中的位运算符。

表 2-8　C#位运算符

运算符	功能	说　　明
&、&=	与	将两个整数按位与，当两个数的对应位都为1时，结果中该位为1，否则为0。后者将结果赋值到左操作数
|、|=	或	将两个整数按位或，当两个数的对应位都为0时，结果中该位为0，否则为1。后者将结果赋值到左操作数
~	取反	将整数按位取反，如果整数对应位为1，则结果中对应位为0，否则为1
^、^=	异或	将两个整数按位异或，当两个数的对应位相同时，结果中该位为0，否则为1。后者将结果赋值到左操作数
<<、<<=	左移	将左操作数按位左移N（右操作数）位。后者将结果赋值到左操作数。后者将结果赋值到左操作数
>>、>>=	右移	将左操作数按位右移N（右操作数）位。后者将结果赋值到左操作数。后者将结果赋值到左操作数

示例代码 2-22 演示位运算符的使用，从中可以看出，位操作符的操作数可以常量，也可以是变量。实际开发过程中，位操作常用在位屏蔽、标记状态等功能中。

示例代码 2-22

```
byte val = 0x81;
System.Console.WriteLine("0x81 >> 4 = 0x" + (val >> 4).ToString("X"));
                                                   //以 16 进制打印
System.Console.WriteLine("0x81 << 4 = 0x" + (val << 4).ToString("X"));
System.Console.WriteLine("0x81 & 0x80 = 0x" + (val & 0x80).ToString("X"));
System.Console.WriteLine("0x81 | 0x80 = 0x" + (val | 0x80).ToString("X"));
System.Console.WriteLine("0x81 ^ 0x80 = 0x" + (val ^ 0x80).ToString("X"));
System.Console.WriteLine("~0x81 = 0x" + ((byte)~val).ToString("X"));
val &= 0x81;
```

示例代码 2-22 的输出如下所示。

```
0x81 >> 4 = 0x8
0x81 << 4 = 0x810
0x81 & 0x80 = 0x80
```

```
0x81 | 0x80 = 0x81
0x81 ^ 0x80 = 0x1
~0x81 = 0x7E
```

2.3.6　用条件运算符判断条件

除了前面几节介绍的常用运算符外，C#还支持一些比较少用的运算符，本节将介绍条件运算符“? :”的使用。条件运算符“? :”是三元运算符，可以根据给定条件进行动态取值，它的定义如下：

```
Condition ? expTrue : expFalse;
```

其中，Condition 是一个布尔类型的表达式，表示选择条件。expTrue 和 expFalse 都为某种类型的表达式，它们的值都可以被赋值到指定的变量中。如果 Condition 为 true，则表达式的值为 expTrue 的值，否则表达式的值为 expFalse 的值。

值得注意的是 expTrue 和 expFalse 有且只有一个表达式被执行，它实际上是简单的 if…else 语句的缩写，所以 expTrue 和 expFalse 不宜食用复杂的表达式和代码段，它们也不能是代码段。

如示例代码 2-23 所示，由于 val1 小于 val2，所以第一个语句中，条件不成立，所以打印后一个字符串。第二个语句中，条件成立，打印前一个字符串。

示例代码 2-23

```
int val1 = 10, val2 = 20;
System.Console.WriteLine((val1 > val2) ? "1----val1 > val2" : "1----val1
< val2");  //打印结果
System.Console.WriteLine((val1 < val2) ? "2----val1 < val2" : "2----val1
> val2");
```

示例代码 2-23 的输出如下所示。

```
1----val1 < val2
2----val1 < val2
```

技巧： 合理使用“? :”运算符可以大大简化代码的行数，使代码可读性更好，但是不要使用复杂的表达式作为 Condition、expTrue 和 expFalse。

2.4　函　　数

函数是编程语言中的又一个基本要素，函数是一段具有特定功能的代码段，它是软件实现功能划分和代码重用的基础。本节将详细介绍 C#中如何编写函数。

2.4.1　定义和使用函数

在 C#中，函数是实现某种特定功能的代码段，它具有明确定义的输入和输出，函数就是通过代码对输入参数进行计算，得到输出结果的代码段。C#中的函数包括函数头和函数

体两部分，在使用之前函数一定要先定义。函数定义的格式如下：

```
[AccessAttr] [static] ReturnType FuncName([ref|out] T1 Para1, [ref|out] T2
Para2,…, [ref|out] TN ParaN)
{
FuncStatements;
}
```

其中，accessAttr 表示函数的可访问性 public、private 等，表示该函数在类外部是否可以访问，更多详细信息见第 3 章。static 关键字表示函数是否为静态函数，为可选项，更多细节参看第 3 章。ReturnType 表示函数返回值的数据类型，可以是任何已经存在且可访问数据类型，如果没有返回值，用 void 表示。圆括号“( )”内是函数参数列表，每个参数都包括参数类型和参数名，参数类型（T1 到 TN）可以是任何已经存在且可访问的数据类型。参数名称（Para1 到 ParaN）是参数名称，命名规则和变量相同，它们最终也将在函数中作为已经存在的变量使用。ref 和 out 是可选修饰符，表示参数是否为引用传递。大括号“{}”是函数体，它可以是任何有代码，也可以是空语句。

函数执行是按照函数体代码依次执行，直到代码执行完，函数就结束。但是更常用的是通过 return 语句退出函数。return 语句有两个作用：退出函数和指定函数返回值。格式如下：

```
return [resultexp];
```

其中，return 是关键字，表示函数到这里退出。resultexp 是计算函数返回值的表达式，它的数据类型应该与函数返回值类型一致。如果函数没有返回值，则一定不要 resultexp，相反如果函数具有返回值，那 resultexp 也是必须的。

如示例代码 2-24 所示，public 是访问修饰符，static 表示函数为静态函数。VoidFunc 是一个没有返回值和参数的函数。Increase 具有一个 int 类型的参数，和 int 类型的返回值，从函数体可以看出 Increase 计算并返回参数 val 加 1 后的值。Sum 具有两个参数，第一个为 int 类型的 val1，第二个为 float 类型的 val2，并返回 float 类型的结果，同样从 Sum 函数体可以看出它返回两个参数之和。

示例代码 2-24

```
public static void VoidFunc()
{
    //return 1;          //错误 不需要返回值
    return;
}
public static int Increase(int val)
{
    //return 1.0;        //错误 1.0 不是 int 类型
    return val + 1;
}
public static float Sum(int val1, float val2)
{
    return val1 + val2;
}
```

定义函数之后，就可以通过函数名加上圆括号“( )”来调用函数，调用函数需要满足两个基本条件，首先需要具有被调用函数的可见性，将在第 3 章详细介绍，本例中的函数

都是可见的（public）。其次，调用者必须为被调用函数提供类型和顺序完全相同的参数。至于函数的返回值是不是需要处理，则不是必须的。

示例代码 2-25 演示如何调用示例代码 2-24 中定义的函数，从代码及注释中可以看出，错误 1 到错误 5 演示了参数类型和顺序不对应，返回值类型不对应的语法错误。最后一个代码段演示将函数变量、表达式、函数返回值等作为函数参数的使用方法。

示例代码 2-25

```
static void Main(string[] args)
{
    //int i = VoidFunc( );  //错误 1，因为 VoidFunc 没有返回值
    //VoidFunc(2);          //错误 2，因为 VoidFunc 不需要参数
    VoidFunc( );

    //Increase(1.0f); //错误 3，因为 1.0 不是 int 类型，Increase 需要 int 类型参数
    Increase(1);            //正确调用，但不处理返回值
    int val1 = Increase(2); //正确调用，并将返回值保存在 val1 中，完成后 val1=3

    //Sum(1.0, 2.0f);       //错误 4，第一个参数 1.0 不是 int 类型
    //Sum(2.0f, 1);         //错误 5，第一个参数和第二个参数顺序颠倒
    Sum(1, 2.0f);           //正确调用，但不处理返回值
    float val2 = Sum(1, 2.0f); //正确调用，将返回值保存在 val2 中，完成后 val2=3.0

    int i = 1, j = 2;
    val1 = Increase(i);     //将变量作为参数，完成后 val1=2
    val1 = Increase(i + j); //表达式作为参数，完成后 val1=4
    val2 = Sum(i, 2.0f);    //变量作为参数，完成后 val2=3.0
    val2 = Sum(Increase(i), 2.0f); //函数表达式作为参数，完成后 val2=4.0
}
```

2.4.2　了解 Main()函数

在 C#程序中，有一个唯一的程序入口函数——Main()函数，操作系统在启动程序时，第一次调用的就是 Main()函数。C#中一切都是类，Main()函数也是类的一个成员函数，但是 Main()函数必须是静态（static）和公开（public）的。

Main()函数包含一个可选的 string[]类型参数，该参数包含了启动程序时的命令行参数，但是不包括程序名。另外，C#中 Main()方法可以声明 void 或 int 两种返回类型。

如示例代码 2-26 所示，是 Main()函数的几种典型的写法，它们都被声明为 static 类型。值得注意的是，C#中只能具有一种形式的唯一的 Main()方法，所以示例中的四种方法只能有一个同时存在。

示例代码 2-26

```
//带参数，返回 void 的 Main()函数
static void Main(string[] args)
{
    return;
}
//不带参数，返回 void 的 Main()函数
```

```
static void Main( )
{
    return;
}
//带参数，返回 int 的 Main()函数
static int Main(string[] args)
{
    return 0;
}
//不带参数，返回 int 的 Main()函数
static int Main( )
{
    return 0;
}
```

2.4.3　区分值传递和引用传递

在 C#中，函数的参数有两种传递方式：值传递和引用传递。在值传递情况下，函数内部使用的参数只是实际参数的一个拷贝（副本），函数内对参数的修改不会影响到原来的参数本身。在引用传递下，函数内部使用的就是实际参数，函数内对参数的修改直接反映到原来的参数上。

在 C#中默认情况下参数是按照值传递的，对于引用传递需要通过 ref 或 out 修饰符指明，在调用函数时也用 ref 或 out 指明。如示例代码 2-27 演示 ref 和值传递的区别，关于 ref 和 out 的区别在 2.4.4 节介绍。

示例代码 2-27

```
namespace Function
{
    class Program
    {
        static void Main(string[] args)
        {
            int val = 1, refVal = 2;
            System.Console.WriteLine("Before TransPara val = {0}", val);
            System.Console.WriteLine("Before TransPara refval={0}",refVal);
            TransPara(val, ref refVal);  //val 为值传递，refVal 为引用传递
            System.Console.WriteLine("After TransPara val = {0}", val);
            System.Console.WriteLine("After TransPara refval = {0}",refVal);
        }
        static void TransPara(int val, ref int refVal)
        {
            val ++;
            refVal ++;
            System.Console.WriteLine("In TransPara val = {0}", val);
            System.Console.WriteLine("In TransPara refval = {0}", refVal);
        }
    }
}
```

通过示例代码 2-27 可见，val 在 TransPara 调用前后都为 1，而在 TransPara 中则为 2，其实是副本的值为 2。而 refVal 在调用前为 1，调用后则变成 3，和在 TransPara 中的值相等。

```
Before TransPara val = 1
Before TransPara refval = 2
In TransPara val = 2
In TransPara refval = 3
After TransPara val = 1
After TransPara refval = 3
```

注意：示例代码 2-27 就是一个典型的 C#程序结构，Program 是程序的启动类，它的成员 Main()函数是入口函数。

2.4.4　区分 ref 和 out 关键字

在 C#中，ref 和 out 关键字都修饰参数，表示参数是引用传递。它们的区别在于：ref 要求参数在传入之前需要进行初始化，而 out 则不需要参数进行初始化。由于 ref 参数必须初始化再传递，所以在函数内部可以直接使用参数的值。同理，由于 out 参数在传递之前并不保证初始化，所以在函数内初始化该参数之前不能直接使用它的值，传递之前进行的赋值对函数本身也没有实际意义。

示例代码 2-28 中，RefParaFunc()定义了 ref 修饰的参数 refVal，在函数内部可以直接使用 refVal 的值，但是调用的地方必须要先对 refVal 进行初始化，例如错误 1 的代码就不正确。OutParaFunc()定义了 out 修饰的参数 outVal，由于不能保证 outVal 被初始化，所以在函数内部初始 outVal 之前不能直接使用它，例如错误 2 的代码就不正确。

示例代码 2-28

```
public static void UseRefOutPara()
{
    float refVal;
    //RefParaFunc(1.0f,ref refVal); //错误 1，refVal 还没有赋值，不能传递
    refVal = 2.0f;
    RefParaFunc(1.0f, ref refVal);  //正确，refVal 赋值后可以传递

    float outVal;
    OutParaFunc(1.0f, out outVal);  //outVal 可以不赋值直接传递
    outVal = 3.0f;
    OutParaFunc(1.0f, out outVal);
                        //outVal 在赋值之后也可以传递，但是该值不会在函数内使用
}
public static void RefParaFunc(float val, ref float refVal)
{
    float val2 = refVal * val;      //refVal 在进入函数前已经赋值，可以直接使用
    refVal = val2;
}
public static void OutParaFunc(float val, out float outVal)
{
    //float val2 = outVal * val;    //错误 2，outVal 可能没有赋值，不能直接使用
    outVal = 2.0f;
    outVal = outVal * val;          //outVal 在赋值之后可以使用
}
```

技巧：理论上讲，out 和 ref 并没有严格的使用限制。通常，该参数仅仅是作为返回值参数，就用 out 修饰，如果该参数的数据需要参与函数内部运算，则用 ref 修饰。

2.4.5　使用 params 关键字

在一些特定场合中，在定义函数的时候，并不能明确知道函数参数的个数，或者说函数的参数个数是不定的。在 C#中，可以通过 params 关键字修饰函数参数来表示该参数是一个数目可变的参数。由于 params 修饰的参数个数可变，所以一个函数只能有一个 params 修饰的参数，而且该参数只能是参数列表的最后一个。另外，params 修饰的参数必须是参数类型的数组类型。

如示例代码 2-29 所示，函数 CalSum()定义了一个可变的 int[]类型参数 valList，表示 CalSum()函数最后接收零个或多个 int 类型的参数，这些参数将放在 int[]类型的 valList 中并传递到 CalSum()函数中。

示例代码 2-29

```
public static void UseCalSum()
{
    System.Console.WriteLine("CalSum() = {0}", CalSum( ));       //0 个参数
    System.Console.WriteLine("CalSum(1, 2) = {0}", CalSum(1, 2));//1 个参数
    System.Console.WriteLine("CalSum(1, 2, 3) = {0}", CalSum(1, 2, 3));
                                                                 //2 个参数
}
public static int CalSum(params int[] valList)
{
    int sum = 0;
    for (int i = 0; i < valList.Length; i++ )   //遍历所有可变参数，并计算和
    {
        sum += valList[i];
    }
    return sum;
}
```

示例代码 2-29 的输出如下所示。

```
CalSum() = 0
CalSum(1, 2) = 3
CalSum(1, 2, 3) = 6
```

技巧：笔者认为，params 参数和数组类型参数最大的区别在于语法上，在数组中元素个数不多的情况下用 params 可以使代码的可读性更好。但是可变参数的个数较多时，数组为参数就更加好用。

2.5　语　　句

在编程语言中，语句就如同人在日常生活中所说的话，它表达了程序真正想做的事情，前面的例子中也用到了很多语句。本节将详细地介绍 C#中的几种对程序结构进行控制的常用语句。

2.5.1　使用 if…else 跳转语句

日常生活中总是会遇到很多选择，编写程序也一样，通常需要进行条件判断，如果条件成立执行一段代码，条件不成立则执行另外一段代码，这一类语句通常可以叫做条件语句（或选择语句）。C#提供两种条件语句：if…else 和 switch，本节将详细介绍 if…else 语句。

C#中，if 语句进行条件判断时，其中 else 分支是可选的，另外，if…else 语句还可以进行嵌套。if 语句的格式如下：

```
if(condition){ TrueStatements; }
[else{ FalseStatements; }]
```

其中，conditon 是类型为 bool 的表达式，作为判断条件。当 condition 为 true 时执行代码块 TrueStatements，反之则执行 FalseStatements。分支代码中可以嵌套任何合法的代码，包括 if…else 本身。另外，如果分支代码只有一句，那么用来分隔代码块的大括号“{}”可以省略，但是笔者强烈建议保留。

如示例代码 2-30 所示，函数 SimpleIf()和 SimpleIfElse()从功能上完全相似，只是前者没有 else 子句，后者有。EmbedIfElse()方法通过嵌套的 if…else 语句计算三个参数 val1、val2 和 val3 的最大值。

示例代码 2-30

```
static void Main(string[] args)
{
    SimpleIf(1);
    SimpleIfElse(11);
    EmbedIfElse(10, 20, 1);
    EmbedIfElse(25, 12, 30);
}
static void SimpleIf(int val)
{
    if(val > 10)                //单个 if 语句
        System.Console.WriteLine("SimpleIf(): val > 10");
    System.Console.WriteLine("SimpleIf(): val <= 10");
}
static void SimpleIfElse(int val)
{
    if (val > 100)              //大于 100 打印提示
        System.Console.WriteLine("SimpleIfElse(): val > 100");
    else                        //其他，即小于等于 100 打印提示
        System.Console.WriteLine("SimpleIfElse(): val <= 100");
}
static void EmbedIfElse(int val1, int val2, int val3)
{
    int maxVal = 0;
    if(val1 > val2)             //如果 val1 大于 val2
    {
        if(val1 > val3)         //如果 val1 大于 val3
            maxVal = val1;
        else                    //如果 val1 小于等于 val3
            maxVal = val3;
```

```
    }
    else                        //如果 val1 小于等于 val2
    {
        if(val2 > val3)         //如果 val2 大于 val3
            maxVal = val2;
        else                    //如果 val2 小于等于 val3
            maxVal = val3;
    }
    //打印出最大值
    System.Console.WriteLine("Max value of ({0}, {1}, {2}) is {3}", val1,
    val2, val3, maxVal);
}
```

示例代码 2-30 的输出如下：

```
SimpleIf(): val <= 10
SimpleIfElse(): val <= 100
Max value of (10, 20, 1) is 20
Max value of (25, 12, 30) is 30
```

理论上讲，if 语句的嵌套深度没有限制，但是随着嵌套层次的增加，代码的复杂度增加，阅读性和维护性下降，更加容易出错，所以一般限制在 3 层以下。如果必须多层嵌套，则可以考虑用函数来分离功能。另外，判决条件表达式不宜太复杂，通常用一个名副其实的 bool 变量来保存复杂的判决条件，增强代码的可读性。

2.5.2　使用 switch 开关语句

实际开发中，常需要判断某个变量是否为一系列的特定值，对于每个值都有特定的处理，而其他值则统一处理或不处理。C#提供 switch 语句实现这种具有并列关系条件的选择执行。switch 语句的定义如下：

```
switch(selectVal)
{
    case val1:
        states1;
    …
    case valn:
        statesn;
    [default:
        defaultStates;]
}
```

其中，selectVal 表示用作选择条件的表达式，它应该是一个整数、字符串和枚举类型。val1 到 valn 都是常量，表示 n 个可选开关。states1 到 statesn 是对应可选开关匹配后需要执行的代码块，必须以 break 或 return 语句结束。default 开关在是没有任何开关匹配上时执行，为可选。

switch 语句在执行时，将 selectVal 依次与 val1 到 valn 进行匹配，如果匹配上则执行对应开关对应的代码块，直到遇到 break 语句退出 switch 语句块，或者遇到 return 退出函数。示例代码 2-31 中，通过 switch 语句根据输入的 Answer 值打印出特定的中文。

示例代码 2-31

```
public enum Answers
```

```
{
    Yes,
    No,
    Right,
    Wrong,
}
static void Main(string[] args)
{
    SwitchFunc(Answers.Yes);
    SwitchFunc(Answers.Wrong);
    SwitchFunc((Answers) 5);         //不正确的 Answers 枚举值
}
static void SwitchFunc(Answers asw)
{
    switch (asw)                     //通过 switch 语句选择 asw 的值
    {
        case Answers.Yes:
            System.Console.WriteLine("是的");
            break;
        case Answers.No:
            System.Console.WriteLine("不是");
            break;
        case Answers.Right:
            System.Console.WriteLine("正确");
            break;
        case Answers.Wrong:
            System.Console.WriteLine("错误");
            break;
        default:
            System.Console.WriteLine("未知");
            break;
    }
}
```

示例代码 2-31 的输出如下，根据不同的 Answers 枚举值，打印出对应的中文，如果不认识则打印出“未知”。

```
是的
错误
未知
```

switch 语句的功能完全可以用 if…else 语句实现，但是 switch 实现的代码具有更好的可读性。switch 任何分支的代码块可以包含任意多行代码，而且不需要大括号“{}”。

2.5.3　用 while 和 do…while 循环语句

循环语句是开发语言中最常用的一个语句之一，在 C#中通过 while 和 do…while 语句实现在满足某条件时重复执行指定代码段。while 和 do…while 的格式如下：

```
while(condition)
{
    repeatStatements;
}
do
{
    repeatStatements;
```

```
}while(condition);
```

其中，condition 是一个 bool 类型的表达式，表示执行条件，repeatStatements 表示 condition 为 true 时执行的代码段。该代码段如果只有一行代码可以省略大括号“{}”，但是笔者强烈建议保留。

while 语句每次循环之前都计算 condition 的值，如果 condition 为 true 则执行重复代码段，否则执行 while 后面的代码。do…while 具有类似的功能，但是它是先执行一遍重复的代码段，然后再判断 condition 的值。由此可见，while 语句中重复代码最少执行 0 次，而 do…while 语句中的重复代码最少执行 1 次。

如示例代码 2-32 所示，其中 WhileFunc()和 DoWhileFunc()都是根据传入参数 x 的值循环打印从 x 到 6 之间的数值。WhileFunc()先判断条件 x<6，而 DoWhileFunc()先执行一遍然后再判断条件 x<6。

示例代码 2-32

```
static void Main(string[] args)
{
    WhileFunc(2);
    DoWhileFunc(2);
    WhileFunc(6);
    DoWhileFunc(6);
}
static void WhileFunc(int x)
{
    System.Console.Write("WhileFunc({0}):", x);
    while (x < 6)                                //如果 x<6 则打印 x 的值
    {
        System.Console.Write("{0},", x);     //打印 x 的值
        x++;                                     //改变循环条件 x 的值
    }
    System.Console.WriteLine( );
}
static void DoWhileFunc(int x)
{
    System.Console.Write("DoWhileFunc({0}):", x);
    do                                           //do 语句会先执行一次打印 x 的值
    {
        System.Console.Write("{0},", x);
        x++;                                     //改变打印条件 x 的值
    } while (x < 6);                             //判断是否继续打印
    System.Console.WriteLine( );
}
```

示例代码 2-32 的输出如下所示，从中可以看出，当参数 x 的值小于 6（即条件 x<6 为 true）时，由于 while 和 do…while 具有相同的效果。而当参数 x 的值大于等于 6（即条件 x<6 为 false）时，while 语句执行了 0 次循环，而 do…while 先执行 1 次循环（即打印出了 x 的值）。

```
WhileFunc(2):2,3,4,5,
DoWhileFunc(2):2,3,4,5,
WhileFunc(6):
DoWhileFunc(6):6,
```

🔔**技巧**：while 和 do…while 的选择通常是根据循环代码是否需要先执行一次来判断，如果是，那么用 do…while，否则用 while。在笔者平时的开发工作中，while 的使用概率要远远大于 do…while。

2.5.4　用 for 和 foreach 遍历语句

循环语句通常包括 3 个部分：条件初始化、重复执行代码、改变重复条件。在 where 中，条件初始化是在 while 语句之前实现的。C#提供另外一种循环语句——for 语句，它将条件初始化也集成到同一语句中，格式如下：

```
for(initCondition; condition; modifyCondition)
{
    repeatStatements;
}
```

其中，initCondition 是一个或多个表达式，是判断条件的初始化。condition 是一个 bool 类型的表达式，当 condition 为 true 时，执行 repeatStatements 的代码。每次循环完成后，执行 modifyCondition 代码，对条件进行更新。

如示例代码 2-33 为最典型的 for 语句应用。for 语句执行时，首先执行初始化代码“int i=0;”，然后判断条件“i<6”，如果条件成立执行大括号“{}”内的重复代码，一次循环后，执行更新条件的代码“i<6”，然后再判断条件“i<6”，条件成立，则继续循环，否则 for 语句执行完成。

示例代码 2-33

```
static void ForFunc()
{
   int[] ary = { 1, 2, 3, 4, 5, 6 };
   for (int i = 0; i < 6; i++)          //打印数组中从 0～5 共 6 个元素
   {
      System.Console.WriteLine(ary[i]);
   }
}
```

由此，可以见 for 语句的效果和 while 语句一样，只是它的代码看起来更加简洁，可读性更好一点。初始化代码和修改条件的代码还可以是多个用“,”分隔的语句，也可以是空语句。如代码：for(int i=0, j=1; i<6; i++, j++);同时初始化 i 和 j，同时更新 i 和 j。

除了 for 语句之外，foreach 是 for 语句的另外一个变种，可以用来循环遍历一个集合（如：数组）中的所有元素，而不使用示例代码 2-33 所示的 for 语句形式。foreach 语句的格式如下：

```
foreach(Type ele in eleList)
{
    repeatStatements;
}
```

其中，Type 是集合中元素的数据类型，ele 是 foreach 语句内部用来表示每次循环从集合中提取的元素变量，eleList 是元素集合。foreach 语句依次从 eleList 中提取元素，每提取

到一个元素就执行一次循环代码。如果 eleList 中没有元素，则循环代码将不会执行。

如示例代码 2-34 所示为典型的 foreach 语句的应用例子，首先定义一个 int 数组 ary，并初始化它的值，然后用 foreach 语句从 ary 中提取元素，每次提取的元素保存在 int 类型变量 val 中。

示例代码 2-34

```
static void ForeachFunc()
{
    int[] ary = { 1, 2, 3, 4, 5, 6 };
    foreach (int val in ary)//打印数组 ary 中的所有元素，foreach 依次遍历这些元素
    {
        System.Console.WriteLine(val);
    }
}
```

技巧： 当重复执行的代码为一行语句时，for 和 foreach 都可以省略大括号“{}”，但是笔者建议保留以提高代码可读性。

2.5.5　用 break 和 continue 控制循环

很多时候，在执行循环语句时，需要中途退出循环或退出本次循环，在 C#中可以在循环代码中使用关键字 break 和 continue 来实现这样的功能。break 语句退出它所在的上一层循环（while、do…while、for、foreach）的所有循环，continue 语句则是退出它所在的上一层循环的本次循环，继续下一次循环。

如示例代码 2-35 所示，第一个 foreach 遇到大于 6 的元素时用 break 语句退出整个循环，实际上是打印出数组 ary 中从第 0 个元素开始的不大于 6 的连续元素。第二个 foreach 语句在遇到大于 6 的元素时用 continue 退出本次循环，继续下一次循环，实际上是打印出数组 ary 中所有不大于 6 的元素。

示例代码 2-35

```
static void BreakContinueFunc()
{
    int[] ary = { 1, 3, 4, 7, 1, 8, 6 };
    System.Console.Write("Break:");
    foreach (int val in ary)
    {
        if (val > 6)
            break;
        System.Console.Write("  {0}", val);
    }
    System.Console.Write("\nContinue:");
    foreach (int val in ary)
    {
        if (val > 6)
            continue;
        System.Console.Write("  {0}", val);
    }
}
```

示例代码 2-35 的输出如下所示，可以看出 break 是退出了整个 foreach 循环，而 continue 只是退出了本次循环，而继续 foreach 的下一次循环。

```
Break:  1  3  4
Continue:  1  3  4  1  6
```

注意：当出现循环嵌套时，break 和 continue 只是退出它们直接的上一层循环，而不是退出嵌套在一起的所有循环。

2.6　小　　结

C#是微软公司与.NET 平台同步推出的面向对象的高级编程语言，它是为.NET 量身打造的，随着.NET 的发展和改进，C#语言也发展到了如今的 C# 4.0 版本，在保持简单易懂等优点的情况下，功能得到了尽可能的增强。

本章作为本书基础章节，也是本书所有知识和例子程序的基础，首先介绍.NET 框架的基础知识及包括垃圾回收机制、公共语言规范在内的重要概念。然后，从编程语言的角度，全面地介绍了 C#作为开发语言的基础知识，包括变量、数据类型、表达式、函数、语句等。通过本章的学习，读者应该掌握以下知识点：

- 什么是.NET？
- .NET 框架和.NET 类库是什么关系？
- 垃圾回收机制如何工作？
- 公共语言规范是什么？在.NET 中如何实现跨语言编程？
- C#应用程序的基本结构如何？
- C#具有哪些数据类型？
- C#具有哪些主要的运算符？
- 如何编写和使用 C#函数？
- C#具有哪些条件语句、循环语句？

第 3 章　C#类和接口

在今天，一门高级程序语言必不可少的一个特性是面向对象。C#也支持面向对象开发，C#通过类、接口等方式全面支持面向对象技术，支持访问性、继承、重载等高级特性，甚至可以说在 C#中所有数据都是对象，包括应用程序本身。本章将详细介绍 C#类和接口相关知识。

3.1　类 和 对 象

类（class）是 C#实现面向对象的最基础的技术，类是一种自定义数据类型。通过定义类，开发人员可以操作各种数据，本节将详细介绍类的基础知识。

3.1.1　区分类和对象

面向对象（Object Oriented，OO）是 20 世纪 90 年代软件开发的主流思想，它是一套指导软件开发工作的理论和思想，并从中总结出一套设计软件和开发软件的方法。面向对象提供了一种对现实世界进行抽象的方法，它的根本就是将事物按照某些共性进行分类，再在同类的基础上实现各自的特性。这样可以在最大程度上提高软件的重用性，并且让软件的思路和逻辑更加清晰。

任何软件开发都是通过某种方法进行数据的处理（逻辑运算、显示、存储等），面向对象思想也是如此。面向对象包含以下几个基本的概念。

- 类：作者认为，类是指事物的共性，这些共性体现了一类事物共有的特性。现实生活中，同一个事物通常同时包含着不同类的共性，所以从不同角度、不同层次来看，它将属于不同的类。比如小狗的名字叫“旺财”，它是一个实实在在存在的个体，从物种来说，它属于“狗”这个类，而从其他方面看，它也属于“宠物”这个类。这里“狗”和“宠物”就分别是两个不同的类，表示不同的共性。
- 对象：对象就是符合某一类共性的具体事物（实例），一个类肯定会有很多对象，否则它也就没有共性可言。比如前面说到的小狗“旺财”就是一个实例，它既是“狗”的实例，又是“宠物”的实例。具体属于哪一个，就需要在实际的应用场合中进行区分。
- 类的数据：类虽然是抽象的，但它同样包含数据，对类的处理往往也是对它的数据进行处理。类的数据分为两类，即属于整个类的数据、个别实例都有但各不相同的数据。比如“狗”是表示所有狗这种动物共性的类，“狗的所有种类”就是属于“狗”这个类的数据，因为就每条狗而言，狗的种类就是那么几种，不会因

为不同的狗而改变。而“狗名”、“颜色”等则是所有“狗”都有的，但是每条狗都不相同，比如“旺财”是“黑色”，“阿虎”是“灰色”等。

- 类的行为：类的行为定义了类共性中的行为部分，定义了如何访问类的数据，如何使用这个类。比如，“狗”这个类可能会有一个方法“摆尾”来向自己的主人表示亲热，还可以有方法“睡觉”来开始睡觉。

在类和对象的基础上，可以非常轻松地理解和实现面向对象思想中的 3 个重要特点。

- 封装：封装是指类把具体的数据访问、行为实现都封装在对外接口中，达到外的数据隐藏和对类本身的封装，外界不能也不用知道类的内部实现。这通过类成员的可访问性来实现，本章将会介绍。
- 继承：继承（派生）是指一个类（子类）可以从另外一个类（父类）继承，“狗”可以从“动物”继承，因为“狗”是“动物”的特例。“军犬”可以从“狗”继承，依次类推。
- 多态：前面讲从不同的角度看一个事物，它会属于不同的类，要完整地表示这个事物就需要同时从多个类继承，让它拥有其父类的共性，这就叫多态。

C#中，通过 class、interface、public、private 等数量不多的关键字非常全面地支持了面向对象技术，本章后面几节将重点介绍类（class）和接口（interface）的基础知识。

3.1.2　定义和使用类

类就是一个开发人员自定义的数据类型，用来表示实际或抽象的一类事物的共性。在使用类之前，首先需要定义类及类的成员。在 C#中，通过 class 关键字来定义一个类，它的常用格式如下：

```
[AccessPara] class className
{
   CalssBody;
}
```

其中，AccessPara 是支持该类的可访问性修饰符，包括 public、private 等。className 是类的名称，它的命名规则和结构体的命名规则一样，同样在同一个命名空间下不能出现相同的类名称。用大括号“{}”括起来的是类的实现部分，这里通常包括类的成员，如字段、属性、方法、事件等，这些将在后面几节详细介绍。

在 C#中，一个类肯定是定义在某个命名空间或类下面，定义之后这个类就属于该命名空间或对应的类，通过该命名空间或类的路径和类名共同组成了新的类的全名，通过这个全名就可以访问新的类。例如，在示例代码 3-1 中，定义了类 Dog，它属于命名空间 DogNS，用 public 修饰表示类 Dog 是公开的，可以在 DogNS 之外的其他命名空间下访问它，这里 Dog 类的全名是 DogNS.Dog。同样地，ClassOfDog 是 Dog 类下面的一个类，它属于 Dog 类，所以 ClassOfDog 的全名是 DogNS.Dog.ClassOfDog。

示例代码 3-1

```
namespace DogNS
{
   public class Dog
```

```
    {
        public class ClassOfDog
        {}
    }
}
```

🔔注意：在 C#中，类名是大小写敏感的，即 clsA 和 clsa 是两个不同的类，但是为了避免出现混淆，强烈建议不要在同一命名空间下出现只有大小写区别的类名。而且，尽量为类取一个具有实际意义，且能表示该类特征的名称。

C#中，在定义类之后，通常有两种方式访问这些类，第一种是通过完整的名称指定类名称，如 DogNS.Dog，第二种是先用 using 语句引用类的部分或全部命名空间，然后使用类相对于被引用命名空间的相对名称来访问。如果在使用“using DogNS;”的情况下，则直接使用 Dog 表示 DogNS.Dog 类。

在 C#中，所有类变量都是一个指向具体对象的引用，类似于 C++中的指针，通过这些引用可以访问对象的成员和方法等，引用都可以为空（null）表示不指向任何对象。如果某个引用为 null，则通过它访问任何成员和方法都会产生异常，所以在使用引用之前首先要为它赋值，即让它指向一个已经存在的对象或新建的对象。在 C#中，通过赋值运算符“=”来进行引用赋值，通过 new 关键字来创建一个新的对象，类对象将一直存在于内存中，直到所有的指向它的引用都退出作用域，它占用的内存才会被垃圾回收机制回收。

如示例代码 3-2 所示，由于先通过语句“using DogNS;”引用了命名空间 DogNS，在变量 dogA 的定义中才可以在 new Dog()中用 Dog 直接访问类 DogNS.Dog。同时，用 new 关键字创建 Dog 类的对象，并赋值到引用 dogA，此时 dogA 指向 new 出来的 Dog 对象，该对象引用计数器变为 1。然后赋值运算符将 dogA 赋给 dogB，使得 dogB 也指向这个对象，此时该对象的引用计数器变为 2。当函数 Main()退出后，dogA 和 dogB 都退出作用域，该对象的引用计数器变为 0，然后很快就被垃圾回收机制回收。

示例代码 3-2

```
using DogNS;              //使用命名空间 DogNS
namespace DefineClass
{
    class Program
    {
        static void Main(string[] args)
        {
            DogNS.Dog dogA = new Dog();
                          //用 new 关键字创建一个 Dog 对象，并赋值到引用 dogA
            Dog dogB = dogA;
                          //将 dogA 所指向的对象赋值到 dogB，此时它们指向同一个对象
            DogNS.Dog.ClassOfDog codA = new Dog.ClassOfDog( );
            Dog.ClassOfDog codB = codA;
        }
    }
}
```

🔔注意：在示例代码 3-2 中，要访问 ClassOfDog 类也需要通过 Dog.ClassOfDog 来访问，因为在 C#中 using 语句只引用命名空间，而不能引用类，如“using DogNS.Dog;”是有语法错误的。

3.1.3　定义类的成员

3.1.2 节中定义的类都没有任何成员，所以它们并不能解决实际问题，要让类具有开发所需要的功能，就要为它添加恰当的成员。在 C#中，通常可以为类添加 4 种成员，即字段、属性、方法和事件，关于类的事件，将在后面详细介绍，本节重点介绍前 3 种。

1．类的字段（Field）

类的字段表示类所包含的数据，这些数据通常可以完整地描述这一类事物在该分类上的共同特性，如狗有颜色、品种、重量、名称等数据。在 C#中，字段的定义和函数中的变量定义非常相似，格式如下：

```
[AccessPara] [static] [readonly] DataType fieldName;
```

其中，AccessPara 表示可访问性，public 为公开，private 为私有（即只在类内部可访问）。关键字 static 表示该成员是否为静态成员，通常一个字段都不是静态成员。关键字 readonly 表示该字段是否为只读字段，如果该字段为只读，那么它只能在构造函数或声明的时候进行唯一一次赋值。DataType 是任何可访问的数据类型，包括类本身。fieldName 是该字段的名称，命名规则和变量的命名规则一样。

定义了字段之后，可以通过域运算符“.”来访问类的字段，格式为 ObjectName.FieldName，表示对象 ObjectName 的字段 FieldName。另外，任何一个类内部都可以通过 this 关键字访问当前对象自身，如果 this.FieldName，表示访问自己的字段 FieldName，this 在属性和方法中十分常用。

在示例代码 3-3 中，根据狗的一些共性，为 Dog 类添加几个字段，string 类型字段_Name 表示 Dog 的姓名，int 类型字段_Weight 表示 Dog 的体重，枚举 DogVarieties 类型是 Dog 类的子类型，表示狗的品种，同时字段_Variety 表示 Dog 的品种。这样，Dog 类就可以比较清楚地体现出狗这种动物的共有特征，通过 new 关键字创建出 Dog 类的实例就可以表示具体的某一条狗。

示例代码 3-3

```
public class Dog
{
    //枚举类型，表示狗的品种
    public enum DogVarieties
    {
        HaBaGou,
        HuDieQuan,
        LangQuan,
        HaQiShi,
        Other,
    }

    //名称，默认为"旺财"
    public string  Name = "旺财";
    //体重，没有初始值
    public int _Weight;
```

```
    //品种，默认为 Other（其他）
    public DogVarieties  Variety = DogVarieties.Other;
}
-----------------------------------------------------------------------
static void Main(string[] args)
{
    DogNS.Dog aDog = new Dog(); //用 new 关键字创建一个 Dog 对象，并赋值到引用 dogA
    System.Console.WriteLine("{0} 的品种是 {1},体重为 {2} kg。", aDog._Name,
    aDog._Variety, aDog._Weight);
    aDog._Name = "阿福";
    aDog._Variety = Dog.DogVarieties.HaBaGou;
    aDog._Weight = 20;
    System.Console.WriteLine("{0} 的品种是 {1},体重为 {2} kg。", aDog._Name,
    aDog._Variety, aDog._Weight);
}
```

实例代码 3-3 中，字段_Name 和_Variety 都在定义时给定了初始值，而_Weight 没有指定初始值，则 C#编译器自动将它初始化为 0。在 Main()函数中，aDog 是 Dog 类型的引用，通过 aDog._Name 可以获取和修改它所指向对象的姓名。示例代码 3-3 的输出如下所示，可见 aDog 的值被成功获取和设置。

```
旺财 的品种是 Other，体重为 0 kg。
阿福 的品种是 HaBaGou，体重为 20 kg。
```

2．类的方法

类的方法通常是提供一系列处理数据的方法，每个方法实现各种特定的功能。在类中定义方法与前面介绍的语法完全一样，只是这些方法的目的和类本身结合更加紧密。如示例代码 3-4 所示，为 Dog 类添加一个名为 GetDogString()的方法，该方法通过 this 关键字访问当前对象的_Name、_Weight 等字段的值，并返回它们的字符串格式。

示例代码 3-4

```
public class Dog
{
    public string GetDogString()
    {
        return string.Format("{0} 的品种是 {1},体重为 {2} kg。", this._Name,
        this._Variety, this._Weight);
    }
}
-----------------------------------------------------------------------
static void Main(string[] args)
{
    DogNS.Dog aDog = new Dog(); //用 new 关键字创建一个 Dog 对象，并赋值到引用 dogA
    System.Console.WriteLine(aDog.GetDogString());
    aDog._Name = "阿福";
    aDog._Variety = Dog.DogVarieties.HaBaGou;
    aDog._Weight = 20;
    System.Console.WriteLine(aDog.GetDogString());
}
```

示例代码 3-4 的输出如下：

```
旺财 的品种是 Other，体重为 0 kg。
```

```
阿福 的品种是 HaBaGou，体重为 20 kg。
```

3．类的属性（Property）

从示例代码 3-3 中可以看出，如果将类的字段声明为公开，外部代码直接修改类的字段值，那么类内部的数据将得不到任何保护。比如 Main()函数为 aDog._Weight 设置一个不合逻辑（如：2000）的值，这会让 aDog 内部的数据变得很不正常。要解决这个问题，就需要限制直接修改字段，提供一种间接修改类的字段方式。

在 C#中，属性是一种解决这个问题的机制。类的属性和字段在使用上非常相似，都是直接通过域运算符“.”来访问，它从表面上看很像一个字段。定一个格式如下：

```
[AccessPara] [static] DataType propName
{
   get { ReadStatements }
   set { SetStatements }
}
```

其中，DataType 是任意可访问的数据类型，propName 是属性名称，命名规则和字段命名规则一样。get 和 set 关键字分别表示读取和设置，它们分别包括读取和设置该属性的代码。get 和 set 必须至少存在一个，如果只有 get 表示该属性为只读，如果只有 set 则表示该属性为只写，get 和 set 同时出现，表示该属性为可读写。

事实上，只写属性是很少见的，因为可写属性完全可以用方法来实现，用属性显得多余。属性实质上也是类的方法，只是语法上变得特殊罢了。get 是原型为“DataType get_propName();”的方法，而 set 则是原型为“void set_propName(DataType value);”的方法，所以属性的 set 子句隐式包含一个参数 value，它包含了要设置的值。

如示例代码 3-5 所示，将字段_Name 和_Weight 设置为 private 防止外部直接读取或修改这个字段的值，提供属性 Name 和 Weight 间接访问它们的值，在属性 Weight 中对设置的值进行合法性检查，如果不在 0～50 之内则不合法，采用默认值 20。

示例代码 3-5

```
public class Dog
{
   //枚举类型，表示狗的品种
   public enum DogVarieties
   {
      HaBaGou,
      HuDieQuan,
      LangQuan,
      HaQiShi,
      Other,
   }

   //名称，默认为“旺财”
   private string _Name = "旺财";
   //属性，获取和设置名称
   public string Name
   {
      get
      {
         return this._Name;
```

```
        }
        set
        {
            this. Name = value;
        }
    }

    //体重，没有初始值
    private int _Weight = 20;
    //属性，获取或设置体重，并对体重设置的值进行保护
    public int Weight
    {
        get
        {
            return this. Weight;
        }
        set
        {
            //如果体重不在 0~50 之内用默认值 20，否则用设置的值
            if((value <= 0) || (value > 50))
            {
                this. Weight = 20;
            }
            else
                this. Weight = value;
        }
    }

    //品种，默认为 Other（其他）
    public DogVarieties Variety = DogVarieties.Other;

    public string GetDogString()
    {
        return string.Format("{0} 的品种是 {1},体重为 {2} kg。", this._Name,
        this.Variety, this. Weight);
    }
}
class Program
{
    static void Main(string[] args)
    {
        DogNS.Dog aDog = new Dog();
                         //用 new 关键字创建一个 Dog 对象，并赋值到引用 dogA
        System.Console.WriteLine("{0} 的品种是 {1},体重为{2}kg。",aDog.Name,
        aDog.Variety, aDog.Weight);
        aDog.Name = "阿福";
        aDog.Variety = Dog.DogVarieties.HaBaGou;
        aDog.Weight = 100;
        System.Console.WriteLine(aDog.GetDogString());
        aDog.Name = "小虎";
        aDog.Weight = 25;
        System.Console.WriteLine("{0} 的品种是 {1},体重为{2}kg。", aDog.Name,
        aDog.Variety, aDog.Weight);
    }
}
```

示例代码 3-5 的 Main()函数中，通过 aDog.Weight 直接获取或设置 aDog 的体重，可见，属性的使用方式表面上和字段一样，而实质上则完全不同。示例代码 3-5 的输出如下：

```
旺财 的品种是 Other,体重为 20 kg。
阿福 的品种是 HaBaGou,体重为 20 kg。
小虎 的品种是 HaBaGou,体重为 25 kg。
```

除了对字段的数据进行保护之外，类的属性还将内部实现和外部接口完全独立。比如字段_Weight 可能会由于精度需要改成 float 类型，而对外接口仍然要用 int 类型时，在示例代码 3-3 中，除了 Dog 内部之外，还需要修改 Main()方法来适应这个改变，而在示例代码 3-5 中，只需要在类 Dog 内部和属性 Weight 的 get 分支进行修改即可。这样就很好地提高接口的重用性和扩展性。

另外，本质上属性是函数，所以它不仅用来封装字段，甚至可以用来进行任何逻辑上的操作，比如返回逻辑上存在但实际不必存在的数据。如 Dog 类增加一个只读属性 WeightInG 来返回以克（g）为单位的体重，这个数据在属性中通过_Weight 字段临时计算产生，如示例代码 3-6 所示。

示例代码 3-6

```
public class Dog
{
   //返回以 g 为单位的体重
   public int WeightInG
   {
      get
      {
         return this._Weight * 1000;
      }
   }
}
```

技巧：笔者强烈建议将类的字段都设置为私有，将字段的访问都通过属性来进行封装，这样会让代码更具扩展性。另外，在命名上将字段和属性分开，可以使代码更加容易维护。笔者的经验是将所有字段名都用下划线前导，如_Weight，对应的属性去掉下划线即可，如 Weight 与_Weight 对应。

3.1.4　控制类成员的可访问性

面向对象的主要目的是数据和行为的封装与隐藏，C#中通过对类、类成员进行可访问性限制来实现数据的隐藏。可访问性即数据是否对外可见及类是否对其他命名空间可见？类成员在类外部是否可见？在 C#中，通过访问性修饰符来修饰类、类成员等，主要包括以下几个修饰符。

- 公共访问权 public：通过 public 关键字修饰的类成员，具有最大的可访问性，对它们的访问可以不受任何限制，外部的类、其他命名空间的类都可以访问这类成员。而通过 public 修饰的类可以被其他命名空间访问。
- 私有访问权 private：通过 private 关键字修饰的类成员，可访问性最低，只有在它们所在的类代码和嵌套类的代码才可以被访问，任何外部区域都不能直接访问它们。如果一个类的成员没有明确可访问修饰符，则默认是 private 类型。private 不能用来修饰类。

- ❑ 内部访问权 internal：通过 internal 关键字声明的类成员，可以被同一个命名空间下的类或成员访问。默认情况下，类没有可访问修饰，认为是 internal 访问性。
- ❑ 受保护访问权 protected：通过 protected 关键字声明的类成员，可以被类本身、该类的嵌套类、从该类派生的任何子类访问。protected 关键字常用来修饰一些需要被子类重载的方法。

另外，C#还支持对同一个类成员使用多个访问修饰符，但是前提是这两个修饰符之间没有冲突，比如，可以对一个类同时使用 internal 和 protected，表示该成员只能被同一个命名空间下的派生类访问。

如示例代码 3-5 中，字段_Weight 就是 private 的，所以在 Main()函数中如果直接访问它就会出现语法错误，在类 Dog 内部就可以任意访问（就如自己可以任意使用自己的私有物品）。而 Weight 属性是 public 的，在 Main()函数中就可以直接访问它，当然在类 Dog 内部也可以任意使用。字段 Variety 就是 public 的，在 Main()方法就直接访问它。

可见，成员可访问性使类的信息得到很好的隐藏，比如示例代码 3-5 中 Dog 类的 private 字段_Weight，就是对外隐藏了体重这一信息的具体实现，防止外部调用者随意这个重要数据。定义一个类成员的可访问性一般从以下几个角度去考虑问题：

- ❑ 如果该成员是类的对外接口，需要让任何类都可以访问，则定义为 public。比如 Dog 类的 GetDogString()方法、Name 属性等，这些数据都是 Dog 类对外的接口。
- ❑ 如果该成员只希望在当前命名空间类可见，则定义为 internal。比如一个人的某些秘密只希望好友知道，那么他的好友圈可以看成是命名空间，而这些秘密则应该是 internal 的。
- ❑ 如果该成员只希望类本身和类的子类访问，则定义为 protected。比如一个父亲只想把一个秘密告诉自己的儿子，那么这个秘密就要定义为 protected。
- ❑ 如果该成员不希望任何其他类可见，只能类本身可见，则定义为 private。这些成员就如一个人永远不为人知的秘密。

3.1.5　重载类的构造函数

现在考虑一个问题，一个类在用 new 关键字新建一个对象之后，这个对象的数据如何进行初始化呢？是调用者显式进行初始化吗？当然这样可以，但是如果调用者忘了，那就容易出错，所以这个方案并不好。

在 C#中，每个类都有一个特殊的方法——构造函数，构造函数在类对象被创建时调用，所以类数据的默认值通常在这里进行赋值。构造函数的函数名和类名相同，不需要明确指定返回类型，因为返回类型默认是当前类。它可以被重载，也可以包含任意多个参数。如示例代码 3-7 所示，类 Dog 有两个不同版本的构造函数 Dog()和 Dog(string name)，前者不接受任何参数，后者接受一个 string 类型参数。如果没有明确为类指定构造函数，C#编译器会自动加上一个默认构造函数，默认构造函数是 public 且不含参数，不包含任何实现代码，等价于示例代码 3-7 中构造函数的第一个版本。

示例代码 3-7

```
public class Dog
```

```
{
    public Dog()
    {}
    public Dog(string name)
    {}
}
```

另外，在定义类字段时也可以给出初始值，就像定义并初始化变量一样。在执行过程中，首先会执行定义时进行的初始化，然后再调用 new 时指定的构造函数版本。这里关于重载的更多介绍见 3.2 节。

如示例代码 3-8 所示，Dog 类包括的字段_Name 在定义时初始化，_Weight 在定义时没有初始化。包括两个构造函数 Dog()和 Dog(string, int)。在 Main()函数中，Dog 对象 aDog 是通过构造函数 Dog()创建并初始化，Dog 对象 bDog 通过构造函数 Dog(stirng, int)创建并初始化。

示例代码 3-8

```
class Dog
{
    //构造函数，等价于默认构造函数
    public Dog()
    {
        System.Console.WriteLine("\tDog(): {0}---{1}", this._Name, this._
        Weight);
        this. Weight = 20;
    }
    //带参数构造函数，初始化姓名和体重
    public Dog(string name, int wt)
    {
        System.Console.WriteLine("\tDog(): {0}---{1}", this._Name, this._
        Weight);
        this. Name = name;
        this. Weight = wt;
    }
    //姓名，定义时进行初始化
    private string _Name = "默认狗名";
    public string Name
    {
        get
        {
            return this. Name;
        }
    }
        //体重，定义时不初始化，默认为 0
    private int _Weight;
    public int Weigth
    {
        get
        {
            return this. Weight;
        }
    }
    //打印当前最新信息
    public void PrintDog()
    {
        System.Console.WriteLine("\tPrintDog(): {0}---{1}", this._Name,
```

```
        this. Weight);
    }
}

class Program
{
    static void Main(string[] args)
    {
        //使用不带参数的构造函数
        System.Console.WriteLine("aDog:");
        Dog aDog = new Dog( );
        aDog.PrintDog( );
        //使用带参数的构造函数
        System.Console.WriteLine("bDog:");
        Dog bDog = new Dog("旺财", 50);
        bDog.PrintDog( );
    }
}
```

示例代码 3-8 的输出如下所示。可见，aDog 的_Name 字段一直没有初始化，使用定义时的初始值，_Weight 在构造函数中进行初始化。bDog 的_Name 和_Weight 都在构造函数中进行了初始化。

```
aDog:
        Dog()：默认狗名---0
        PrintDog()：默认狗名---20
bDog:
        Dog()：默认狗名---0
        PrintDog()：旺财---50
```

3.1.6　提供类的静态成员

有一种类的成员属于整个类，而不是属于某个具体对象，这种成员就是静态成员。在 C#中，通过 static 修饰某个类成员表示该成员为静态成员，可以是静态字段、静态属性和静态方法。静态成员属于整个类，所以它们的访问不需要有类的对象存在，直接通过类名进行访问，格式为：类名.成员名称，同样“.”为域运算符。

静态成员属于整个类，在任何地方修改这个成员，都将体现在该类的所有实例对象中，包括已经存在和新创建的对象。同样，类的静态属性只能访问类的静态字段和静态方法，类的静态方法则只能访问类的静态字段和静态属性，这是因为非静态的属性和方法都有一个隐式的成员 this 在内，对于静态属性和静态方法是根本不存在 this 的。如示例代码 3-9 所示，静态成员只能通过类名访问，通过对象访问静态成员在 C#中不允许。

示例代码 3-9

```
class Dogs
{
    //静态成员，狗的数量
    public static int Count = 0;
    //非静态成员，狗的名称
    public string Name;
    //创建一个狗，数量加 1
```

```
    public Dogs( )
    {
        Count++;
    }
    //静态方法，只能访问静态字段
    public static int GetCount()
    {
        //string nm = this.Name;          //错误，不能访问非静态成员
        return Count;
    }
}
class Program
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Dogs.Count = {0}", Dogs.Count);
        Dogs aDog = new Dogs( );
        System.Console.WriteLine("Dogs.Count = {0}", Dogs.Count);
        //直接通过类名调用静态成员
        Dogs.Count = 5;
        System.Console.WriteLine("Dogs.Count = {0}", Dogs.Count);
        Dogs bDog = new Dogs( );
        //调用静态成员函数
        System.Console.WriteLine("Dogs.GetCount() = {0}", Dogs.Get-
        Count( ));
        //aDog.GetCount( );   //错误，静态成员只能通过类名访问。
    }
}
```

示例代码 3-9 的输出如下，从中可以看出，任何地方对静态字段 Count 的更改（类内部或是类外部）都会反映到所有的类对象中。

```
Dogs.Count = 0
Dogs.Count = 1
Dogs.Count = 5
Dogs.GetCount() = 6
```

3.1.7　添加类的索引器

在 C#中，还提供一种特殊的属性——索引器，通过它可以像使用数组那样使用一个类，这在作为数据集存在的容器类中相当有用。索引器通过关键字 this 来定义，常用格式为：

```
[accessAttr] DataType this[indexType index]
{
    get{}
    set{}
}
```

其中，accessAttr 表示可访问性修饰符，包括 public、private 等。DataType 表示索引器作为一个属性本身的返回值类型，可以是任意可访问的数据类型。this 是关键字，不可修改。indexType 是作为索引的参数的类型，index 是索引参数。this 和“[ ]”一起表示该属性为索引器。

如示例代码 3-10 所示，MyIndexer 类是一个自定义索引器，它保存了一组字符串，定义了一个 int 类型为参数的索引器，返回类型为某个特定的字符串。Get 子句中，如果索引

超出数据范围就返回默认值——“不存在”，否则返回对应索引的字符串。Set 子句中，如果在范围内修改对应索引的字符串，否则将设置的值添加到容器中。

示例代码 3-10

```
class MyIndexer
{
    //定义一个字符串列表，用来保存所有字符串
    private List<string>  StrList = new List<string>();
    public void AddString(string str)
    {
        this._StrList.Add(str);
    }

    //读写的索引器，支持 int 类型的索引
    public string this[int index]
    {
        get     //如果在正确范围内，直接返回，否则返回默认值
        {
            if ((index >= 0) && (index < this. StrList.Count))
                return this. StrList[index];
            else
                return "不存在";
        }
        set     //如果在正确范围内，更新数据，否则直接添加
        {
            if ((index >= 0) && (index < this. StrList.Count))
                this. StrList[index] = value;
            else
                this. StrList.Add(value);
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        //创建并初始化
        MyIndexer myIndex = new MyIndexer( );
        myIndex.AddString("One");
        myIndex.AddString("Two");
        myIndex.AddString("Three");
        //访问在范围内和不在范围内的数据
        System.Console.WriteLine("[0]: " + myIndex[0]);
        System.Console.WriteLine("[5]: " + myIndex[5]);
        //更新在范围内的数据
        myIndex[0] = "OneOne";
        System.Console.WriteLine("[0]: " + myIndex[0]);
        //更新不在范围内的数据
        myIndex[5] = "FourFour";
        System.Console.WriteLine("[3]: " + myIndex[3);
    }
}
```

示例代码 3-10 的输出如下所示，从中可以看出，使用索引器之后使得 MyIndexer 对外

公开的数据访问更加简单和安全，调用者不再担心数组越界的问题。还有一个好处是，这样可以将 MyIndexer 内部的具体实现完全隐藏起来，例如本例中，Main()函数只看到输入 int 索引返回 string 这样一个接口。另外，这种写法也让 MyIndexer 看起来更像一个容器，更方便使用。

```
[0]: One
[5]: 不存在
[0]: OneOne
[3]: FourFour
```

3.2　类 的 继 承

类的一个重要特性就是继承，就是子类从父类继承，子类具有父类拥有的所有成员和功能。通过继承能够更好地实现代码重用，使得程序结构更加简单合理。本节将详细介绍 C#中如何实现类继承。

3.2.1　从父类派生子类

在面向对象理论中，如果 A 的所有特性 B 都具有，而 B 的特性 A 不具备，这样就可以称为 B 是 A，但并非所有 A 都是 B，就如狗是动物，但并非所有动物都是狗。值得注意的是，特性 A 和 B 是指从某个角度去看待事物得到的特性，而不是说这个事物的全部特性。

在 C#中，通过类的继承来实现这种关系，类 Bclass 从类 Aclass 继承而来，那么 Bclass 具有 Aclass 的非 private 所有成员。Bclass 是 Aclass 的子类，Aclass 是 Bclass 的父类。在 C#中，通过冒号“:”运算符定义类的继承关系，格式如下：

```
calss SubClassName : SuperClassName
```

其中，SubClassName 是子类的名称（新的类），SuperClassName 是父类的名称（已经存在的类），冒号“:”表示 SubClassName 从 SuperClassName 继承。其他的地方和直接定义类 SubClassName 一样。如示例代码 3-11 所示，类 Dog 是从类 Animal 继承而来，这里需要注意的是，在定义 Dog 之前，它的父类 Animal 必须已经存在而且可见。

当一个子类从父类继承之后，子类就直接具有父类的所有 public 和 protected 成员，但是 private 成员是父类专有，子类不能直接访问。如示例代码 3-11 中，Dog 类从 Animal 类集成而来，那么 Dog 类就直接具有 Weight 属性和 Shout()方法，并且都是 public 的，同时它还可以访问 Animal 的 protected 字段_Weight，例如 ShowMe()方法中使用它。但是 Dog 类不能访问 Animal 类的 private 字段_PrivateValue。同样，子类从父类继承之后，通常需要添加专有的成员，否则继承就没有意义（因为子类父类相同，直接用父类即可）。如 Dog 类增加 Name 属性和 ShowMe()方法。

示例代码 3-11

```
//父类 Animal
class Animal
{
```

```
    //private 修饰，它的子类不可以访问
    private string  PrivateValue = "private";
    //protected 修饰，它的子类继承得到
    protected int  Weight = 20;
    //pubic 修饰，它的子类继承得到
    public int Weight
    {
        get
        {
            return this._Weight;
        }
        set
        {
            this._Weight = value;
        }
    }
    //pubic 修饰，它的子类继承得到
    public void Shout( )
    {
        System.Console.WriteLine("An animal is shouting...");
    }
}
class Dog : Animal
{
    //子类专有，父类没有
    private string _Name = "旺财";
    //子类专有，父类没有
    public string Name
    {
        get
        {
            return this._Name;
        }
        set
        {
            this. Name = value;
        }
    }
    //子类专有，父类没有
    public void ShowMe()
    {
        //子类可以使用从父类继承得到的 protected 字段：_Name, _Weight
        System.Console.WriteLine("A dog, name is {0}, and weight is {1} ...",
        this. Name, this. Weight);
    }
}
```

在 C#中，可以用父类的引用指向一个子类的对象，如示例代码 3-12 中，Animal 对象 aml 直接指向 Dog 对象 aDog。当一个父类引用指向一个子类对象时，可以通过 is 运算符来判断是否属于某个类型，关于 is 关键字将在 3.2.7 节详细介绍。另外，父类的引用只能访问父类的成员方法，即使它实际上是子类对象。如 aml 虽然指向 Dog 对象 aDog，但它能访问 Shout()但不能访问 ShowMe()，因为 Shout()是 Animal 的成员，但 ShowMe()是 Dog 的成员。要想调用，只能将父类对象强制类型转换为子类类型，如代码((Dog) aml).ShowMe( );就是将 aml 强制转换为 Dog 类，再调用 Dog 的成员 ShowMe()，但是这里

需要保证 aml 真的是一个 Dog（或其子类）对象，否则强制类型转换要产生异常。

示例代码 3-12

```
static void Main(string[] args)
{
    //创建一个 Dog 对象
    Dog aDog = new Dog( );
    //使用 Dog 从 Animal 继承得到的属性 Weight
    System.Console.WriteLine("aDog.Name={0},aDog.Weight = {1}", aDog.Name,
    aDog.Weight);
    //使用 Dog 专有的成员方法 ShowMe()
    aDog.ShowMe( );
    //使用 Dog 从 Animal 继承得到的方法 Shout()
    aDog.Shout( );

    //Dog 是 Animal，所以可以将 Dog 对象直接赋值到 Animal 引用
    Animal aml = aDog;
    //使用 Animal 的成员方法
    aml.Shout( );
    //aml.ShowMe( );          //错误，因为 Animal 不能直接访问子类的方法
    ((Dog) aml).ShowMe( );
                    //正确，先强制转换成 Dog 在使用，因为 aml 本身就是一个 Dog 对象

    Animal bAml = new Animal( );
    Dog bDog;
    //bDog = (Dog) bAml;      //异常，因为 bAml 实际上不是 Dog 对象
    bDog = (Dog) aml;         //正确，因为 aml 本身就是一个 Dog 对象

    System.Console.WriteLine("aml is Animal : {0}", aml is Animal);
    System.Console.WriteLine("aml is Dog : {0}", aml is Dog);
    System.Console.WriteLine("bAml is Animal : {0}", bAml is Animal);
    System.Console.WriteLine("bAml is Dog : {0}", bAml is Dog);
}
```

示例代码 3-12 演示了父类 Animal 和子类 Dog 的使用，输出结果如下，从中可以看出 Dog 类（子类）对象既是 Dog 类型（子类本身），同样也是 Animal 类型（父类）。但是 Animal 对象（父类）就不是 Dog 类型（子类）。

```
aDog.Name = 旺财, aDog.Weight = 20
A dog, name is 旺财, and weight is 20 ...
An animal is shouting...
An animal is shouting...
A dog, name is 旺财, and weight is 20 ...
aml is Animal : True
aml is Dog : True
bAml is Animal : True
bAml is Dog : False
```

3.2.2　重载类的方法

很多情况下，一个类的同一个动作会有多个版本，各版本功能上很相似，但是实现过程有一定的区别，包括输入、输出和处理过程。在 C#中，可以通过函数重载满足这种设计需要，重载是一种 C#特性，它允许一个类具有多个相同名称的方法（函数），但是这些方

法的输入不同，包括参数的类型和顺序。如示例代码 3-13 所示，计算器类 Caculator 包含 4 个不同版本的 Add()方法，其中 1 和 2 是因为参数类型不一样，2 和 3 是因为参数的顺序不一样。而 4 和其他 3 个版本则是参数个数不一样。

在使用重载函数时，只需要按照需要的重载版本传入参数即可，C#编译器会根据传入参数的类型和顺序自动调用对应的重载版本。示例代码 3-13 中，Main()函数分别使用 4 个不同的重载版本，这些版本的定义都已经存在，所以代码没有问题，并且可以正确运行。但是最后一次调用的版本没有定义，所以语法上有错误。

示例代码 3-13

```
class Caculator
{
    //1—int Add(int,int) 版本
    public int Add(int val1, int val2)
    {
        System.Console.WriteLine("Add(int,int): {0} + {1} = {2}", val1, val2,
        val1 + val2);
        return val1 + val2;
    }
    //2—float Add(int,float) 版本
    public float Add(int val1, float f1)
    {
        System.Console.WriteLine("Add(int,float): {0} + {1} = {2}", val1, f1,
        val1 + f1);
        return val1 + f1;
    }
    //3—float Add(float,int) 版本
    public float Add(float f1, int val2)
    {
        System.Console.WriteLine("Add(float,int): {0} + {1} = {2}", f1, val2,
        f1 + val2);
        return f1 + f1;
    }
    //4—int Add(int) 版本
    public int Add(int val)
    {
        System.Console.WriteLine("Add(int): {0} + 1 = {1}", val, val + 1);
        return val + 1;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Caculator calor = new Caculator( );
        //分别调用 4 个不同的版本
        calor.Add(1, 2);
        calor.Add(1, 2.2f);
        calor.Add(2.2f, 1);
        calor.Add(3);
        //calor.Add(1.1f, 2.2f); //错误，没有 Add(float,float)版本的重载
    }
}
```

示例代码 3-13 的输出如下，可以看出，编译器完全可以找到正确的重载版本并正确执行。

```
Add(int,int): 1 + 2 = 3
Add(int,float): 1 + 2.2 = 3.2
Add(float,int): 2.2 + 1 = 3.2
Add(int): 3 + 1 = 4
```

注意：C#中，如果两个函数的参数列表完全相同，只有返回类型不同，是不允许的，这也不是函数重载。因为 C#编译器无法根据返回类型判断一个函数的具体实现。

3.2.3　子类重载父类的虚函数

继承的一个重要作用是抽象出统一的接口，该接口被所有的子类共同拥有和使用，但是同样一个接口，不同的子类的实现可能有所变化，即父类的实现并不能满足所有子类的要求。这时，就需要有一种机制可以让子类重写父类提供的接口，并且通过父类引用能够正确调用子类的实现。在 C#中，虚函数就是这样一种机制。

用 virtual 关键字修饰的成员函数称为虚函数，一个虚函数可以被子类重载，子类中的重载函数用 override 关键字修饰，如果没有显式说明，则默认用 override。它们的格式如下：

```
[accessAttr] virtual [returnType] FuncName(paraList);
[accessAttr] override [returnType] FuncName(paraList);
```

其中，accessAttr 是访问属性，由于被子类重载，所以它必须对子类可见，accessAttr 就必须不能是 private。virtual 和 override 是关键字。子类重载父类的函数，子类中的重载函数和它在父类中的原型必须在定义上相同，包括函数的返回类型、参数类型、参数顺序。

在 C#中，当虚函数被重载后，通过父类引用调用虚函数，实际上调用的是与该引用所指向对象的类型最近的一个实现。另外，在子类中还可以通过 base 关键字显式调用父类的函数实现。子类可以通过 new 关键字隐藏父类定义的接口，在隐藏之后不能通过父类引用调用到该子类对这个函数的具体实现。

如示例代码 3-14 所示，类 Dog 从类 Animal 继承而来，类 HaBaDog 又从类 Dog 继承而来，所以 Dog 和 HaBaDog 都是 Animal，在 Main()函数中可以通过 Animal 引用指向 3 种类型的对象。Dog.Walk()隐藏了 Animal.Walk()的实现，并且通过 base.Walk()显式地调用父类 Animal 中的 Walk()方法。

示例代码 3-14

```
class Animal
{
    //虚函数 Shout()
    public virtual void Shout()
    {
        System.Console.WriteLine("Animal.Shout()......");
    }
    //虚函数 Run()
    public virtual void Run()
    {
        System.Console.WriteLine("Animal.Run()......");
    }
    //虚函数 Walk()
    public virtual void Walk( )
```

```
    {
        System.Console.WriteLine("Animal.Walk()......");
    }
}

class Dog : Animal
{
    //重写 Animal.Shout()
    public override void Shout( )
    {
        System.Console.WriteLine("Dog.Shout()......");
    }
    //重写 Animal.Run()
    public override void Run( )
    {
        System.Console.WriteLine("Dog.Run()......");
    }
    //重写 Animal.Walk()
    public new void Walk( )
    {
        System.Console.WriteLine("Dog.Walk()......");
        base.Walk( );       //显式调用父类的实现
    }
}

class HaBaDog : Dog
{
    //重写 Dog.Shout()
    public override void Shout( )
    {
        System.Console.WriteLine("HaBaDog.Shout()......");
    }
}

class Program
{
    static void Main(string[] args)
    {
        //演示 Animal 的虚函数调用
        System.Console.WriteLine("1---------Animal------------1");
        Animal aml1 = new Animal( );
        aml1.Shout( );
        aml1.Run( );
        //演示 Dog 的虚函数调用
        System.Console.WriteLine("2---------Dog---------------2");
        Animal aml2 = new Dog( );
        aml2.Shout( );
        aml2.Run( );
        //演示 HaBaDog 的虚函数调用
        System.Console.WriteLine("3---------HaBaDog-----------3");
        Animal aml3 = new HaBaDog( );
        aml3.Shout( );
        aml3.Run( );
        //演示 base 在虚函数中的使用
        System.Console.WriteLine("4---------Dog.Walk()--------4");
        aml2.Walk( );
        Dog adog = (Dog) aml2;
        adog.Walk( );
    }
}
```

示例代码 3-14 的输出如下，可以看出，虽然都是通过 Animal 引用访问对象，但是由于 aml1 本身就是 Animal 对象，所以通过 aml1 调用虚函数都是 Animal 类的实现。aml2 本身是 Dog 对象，所以通过 aml2 调用虚函数调用的都是 Dog 类的实现。aml3 本身是 HaBaDog 对象，通过 aml3 调用虚函数 Shout()调用的是 HaBaDog.Shout()，但是由于 HaBaDog 没有重载虚函数 Run()，所以通过 aml3 调用 Run()会调用离 HaBaDog 类最近的父类的 Run()的实现，所以调用了 Dog.Run()。另外，由于 Dog 类隐藏了 Walk()的实现，所以通过 aml2 调用 Walk()只能调用到 Animal.Walk()，因为 aml2 只是 Animal 类引用。而同一个对象，转换成 Dog 引用 aDog 之后，才可以访问 Dog.Walk()方法。

```
1---------Animal-----------1
Animal.Shout()......
Animal.Run()......
2---------Dog--------------2
Dog.Shout()......
Dog.Run()......
3---------HaBaDog-----------3
HaBaDog.Shout()......
Dog.Run()......
4---------Dog.Walk()--------4
Animal.Walk()......
Dog.Walk()......
Animal.Walk()......
```

注意：虚函数的调用是从实际类型的继承树从下往上一层层查找父类中对该函数的实现，调用找到的第一个实现。另外，由于在 C#中属性本身也是函数，所以属性也是可以被重载的。由于有了函数重载，才使得父类定义的接口具有可扩展性，真正成为接口。

3.2.4　区分抽象类和静态类

在 C#中，如果一个类必须被继承，则认为这个类是抽象类。抽象类用关键字 abstract 修饰，抽象类不能被直接实例化进行使用。另外，抽象类可以包含抽象函数，即用 abstract 修饰的函数。抽象函数是虚函数的一种特例，它本身不提供任何具体实现，只是一个函数接口的定义而已，它要求子类必须重载该函数，除非这个子类本身也是抽象类。

如示例代码 3-15 所示，类 Animal 是一个抽象类，它包含虚函数 Shout()和抽象函数 Run()。类 Dog 继承自 Animal，并且不是抽象类，所以它必须重载 Animal 的所有抽象方法，这里是 Run()。而类 Chicken 由于本身也是抽象类，所以它可以不重载 Animal.Run()，但是它的任何一个非抽象子类都必须重载该方法。Main()方法中，前面两条语句试图创建 Animal 和 Chicken 类的实例都是有语法错误的，因为抽象类不能创建任何实例。

示例代码 3-15

```
abstract class Animal
{
    public virtual void Shout()
    {
        System.Console.WriteLine("Animal.Shout()......");
    }
```

```
    public abstract void Run( );
}
class Dog : Animal
{
    //必须重载父类中的抽象函数，除非它本身也是一个抽象类
    public override void Run( )
    {
        System.Console.WriteLine("Dog.Run()......");
    }
}
abstract class Chicken : Animal
{}
-----------------------------------------------------------
static void Main(string[] args)
{
    //Animal aml = new Animal( );   //错误，不能创建抽象类实例
    //Animal aml = new Chikcen( );  //错误，不能创建抽象类实例
    Animal aml = new Dog( );        //正确，Dog 类不是抽象类，可以创建实例
}
```

技巧：一些非常通用的类往往采用抽象类，这样可以从语法上增加程序的结构性。如果一个类是抽象类，并且只包含抽象的属性或方法，那么它的功能和接口就很相似了。

在 C#中，同样可以通过抽象类访问它的静态成员。对于一些只是对一些方法进行打包，不需要任何实例的类而言，可以将它的所有成员都定义为静态（static），同时将类定义为抽象类，这样就只能访问该类的静态成员。另外，C#还支持用 static 关键字修饰该类来达到这样的效果，用 static 修饰的类称为静态类。一个静态类是只能包含静态成员的抽象类。如 System.Math 就是这样的一个典型。

3.2.5　定义密封类

和抽象类相反，密封类则是不允许被继承，这通常是一些比较专用的类，比如计算器类，所有的计算方法都是按照国际约定进行，不需要任何重载。在 C#中，密封类用 sealed 关键字修饰，它不能包含任何虚函数和抽象函数，因为它不能被继承，就没有重载的可能。如示例代码 3-16 所示，类 Caculator 是用 sealed 的密封类，它不能被继承。但是它同样可以被创建实例。

示例代码 3-16

```
sealed class Caculator
{
    static int Add(int val1, int val2)
    {
        return val1 + val2;
    }
    static int Sub(int val1, int val2)
    {
        return val1 - val2;
    }
}
```

3.2.6　全部类的父类 Object 类

在 C#中一切都是类，所有的类构成了一个完整的树状层次结构。在 C#中，有一个特殊的类——Object 类，它是所有类的基类，任何一个类如果没有明确指定其父类，那么它默认就从 Object 类继承。所以前面的 Caculator 类、Animal 类、Program 等实际上都是从 Object 类继承而来。

Object 类作为所有类的基类，它提供了任何类都需要实现的方法和属性，主要包括如下几个。

- bool Equals(object obj)：该方法用来判断当前对象是否和传入对象相同，相同返回 true，否则返回 false。
- int GetHashCode()：该方法返回一个二进制哈希值，用来唯一表示当前对象。在将数据存储到 Hash 表时，可以用它的哈希值作为 Key 值，非常实用。
- Type GetType()：该方法返回当前对象的实际数据类型，比如 System.String、Animal、Caculator 等。
- object MemberwiseClone()：该方法用来为当前对象赋值一个完全一样的副本。
- string ToString()：该方法返回一个表示当前对象的字符串，默认情况下返回当前对象的类型字符串。具体的类根据需要可以重写这个函数来得到不同的值，比如 String 类重写 ToString()返回字符串本身，int 类则重写 ToString 返回当前值的字符串形式。

由于 Object 类是所有类的基类，所以 Object 对象可以用来表示任何对象。如示例代码 3-17 所示。

示例代码 3-17

```
Object obj = new Caculator();
Object obj = new Animal();
Object obj = new Chicken();
```

3.2.7　区分 as 和 is 关键字

在 C#中，每个引用都有自己的类型，但是有很多种情况下，实际上根本不知道某个引用的具体类型，然后根据它们的具体类型进行特定处理。在 C#中，可以通过 is 运算符判断引用的具体类型，它的具体格式如下：

```
refObj is DataType;
```

其中，refObj 是要判断类型的引用名，DataType 是目标数据类型，is 是关键字。如果 refObj 是 DataType 表示的数据类型，该表达式返回 true，否则返回 false。如示例代码 3-18 所示，Dog 继承自 Animal，aml 实际是 Animal 对象，adg 实际是 Dog 对象，可以通过 is 关键字判断 aml 和 adg 的具体类型。

示例代码 3-18

```
class Animal
{}
```

```
class Dog : Animal
{}
class Program
{
    static void Main(string[] args)
    {
        Animal aml = new Animal();
        Dog adg = new Dog( );
        System.Console.WriteLine("aml is Animal----{0}", aml is Animal);
        System.Console.WriteLine("aml is Dog----{0}", aml is Dog);
        System.Console.WriteLine("adg is Animal----{0}", adg is Animal);
        System.Console.WriteLine("adg is Dog----{0}", adg is Dog);
    }
}
```

示例代码 3-18 的输出如下所示，可见，通过 is 可以真正判断出引用的具体类型。

```
aml is Animal----True
aml is Dog----False
adg is Animal----True
adg is Dog----True
```

在实际开发中，判断出引用的具体类型后可以通过强制类型转换获得对应类型的引用，如果上面的代码可以用代码：Dog adog = (Dog)adg；将对象 adg 转换为 Dog 类型并赋值给 adog。但是这里的 adg 必须是 Dog 类，否则这个强制转换就会出现异常。那么有没有办法既判断类型同时又进行正确的转换呢？有，那就是 as 运算符。

在 C#中，as 运算符判断对象类型，并转换成需要的类型引用，格式如下：

```
desObj = refObj as DataType;
```

其中，refObj 是一个任意类型的对象引用，DataType 是目标类型，desObj 是一个 DataType 类型的引用，as 是关键字。如果 refObj 是 DataType 类型，desObj 将得到 DataType 类型的引用，否则为 null。如示例代码 3-19 所示。

示例代码 3-19

```
static void UseAs()
{
    Animal aml = new Animal( );
    Dog adg = new Dog( );

    Dog adog = aml as Dog;
    if(adog == null)
        System.Console.WriteLine("aml 不是 Dog...");
    else
        System.Console.WriteLine("aml 是 Dog...");
    Animal aAml = adg as Animal;
    if(aAml == null)
        System.Console.WriteLine("adg 不是 Animal...");
    else
        System.Console.WriteLine("adg 是 Animal...");
}
```

示例代码 3-19 的输出如下，由于 aml 不是 Dog 类型，所以“aml as Dog”返回的 adog 为 null。而 adg 是 Dog 类型同时也是 Animal 类型，所以“adg as Animal”返回的 aAml 不为 null，而是 Animal 类型的引用。

```
aml 不是 Dog...
adg 是 Animal...
```

3.3　定义和实现接口

在 C#中提供另外一种新的机制——接口，用来定义统一的接口，就好像前面说到的类一样，但是它比类更加抽象，本节将详细介绍接口的相关知识。

3.3.1　定义接口

在前面章节讲到，类的继承可以通过父类定义和实现所有子类的公用功能来使得子类直接具有这些功能，但是在有些特殊情况下，通过父类进行抽象并不理想。看看下面两种情况：

- 多个子类之间只是要用到的方法名称相同，但是方法的实现根本毫不相干。通过父类提供的方法不能提供任何有用的实现，除了方法名，这样会让父类变得没有多少意义。
- 一个子类希望同时从多个父类继承，使得它能具有多重身份。在 C#中，为了避免多重继承带来的复杂性和易错性，并不支持一个类同时从多个类继承。

为了解决前面的问题，C#提供了一种新的机制——接口（Interface）。和类一样，接口定义了一类事物（类）都必须实现的行为，这些行为是方法、属性、事件的集合，它不提供任何具体实现，也不包含任何字段。接口通过 interface 关键字来定义，在其代码块内指定它所包含的具体成员。

在示例代码 3-20 中，定义了一个接口 IRunable，该接口定义了可以跑的任何动物都需要具备的行为。其中定义了一个方法 Run()，和一个读写属性 Speed 表示当前的速度，还包括一个只读属性 Distance 表示当前的路程。

示例代码 3-20

```
interface IRunable
{
    float Distance
    {
        get;
    }
    float Speed
    {
        get;
        set;
    }
    void Run( );
}
```

技巧：虽然理论上说接口的名称是任意的，但是为了不和类引起歧义，接口的名称通常以大写字母 I 为首，如 IMoveable。

接口的成员在命名规则、表达式格式等各个方面都和类完全一样，只是少了具体的实现而已。另外，在接口中所有成员都是 public 的，所以不需要为它们提供可访问性。最后，接口中不能定义字段，只能定义属性、方法和事件。如 IRunable 就定义了属性和方法。

3.3.2　在类上实现接口

在定义了接口之后，就需要实现接口，在 C#中接口可以被类一样作为数据类型使用，可以被特定的类实现。实现接口就是为该接口提供它所定义的所有成员的具体实现，如果一个类实现了接口，那么它同时也是这个接口类型。

在 C#中，实现接口在语法上与从父类继承完全一样，也是通过冒号“:”来表示，如示例代码 3-21 所示，类 Cow 就是实现了 IRunable 接口的类，所以 Cow 的对象也是 IRunable 类型。在 Main()方法中可以直接将 Cow 对象赋值到 IRunable 引用 run，并且可以像使用类那样使用 run。Cow 必须实现 IRunable 所有的成员。

示例代码 3-21

```
class Cow : IRunable
{
    private float  Distance = 0.0f;
    //实现 IRunable.Distance 属性
    public float Distance
    {
        get
        {
            return this._Distance;
        }
    }
    private float _Speed;
    //实现 IRunable.Speed 属性
    public float Speed
    {
        get
        {
            return this. Speed;
        }
        set
        {
            this._Speed = value;
        }
    }
    //实现 IRunable.Run()方法
    public void Run()
    {
        System.Console.WriteLine("一头牛正以 {0} 米每秒的速度奔跑", this.
        Speed);
        this._Distance += this._Speed;
    }
}
class Program
{
    static void Main(string[] args)
    {
        //定义 IRunable 引用，并初始化为一个 Cow 对象
```

```
        IRunable run = new Cow( );
        //直接使用 IRunable，它实际上是 Cow
        run.Speed = 10;
        run.Run( );
        System.Console.WriteLine("1-总路程：{0}", run.Distance);
        run.Speed = 20;
        run.Run( );
        System.Console.WriteLine("2-总路程：{0}", run.Distance);
    }
}
```

示例代码 3-21 的输出如下所示，从中可以看出，实际上 Irunable 引用 run 调用的方法也是它所指向的实际类型提供的方法，这里是 Cow 提供的方法。

```
一头牛正以 10 米每秒的速度奔跑
1-总路程：10
一头牛正以 20 米每秒的速度奔跑
2-总路程：30
```

3.3.3　在类上实现多个接口

在 3.3.1 节中提出了两个问题，第二个问题在 C#中无法用类来解决，但是可以通过接口来实现，因为在 C#中支持一个类同时实现多个接口。当一个类实现多个接口时，多个接口用逗号“,”分开即可，但是该类必须实现所有这些接口定义的成员，这样这个类就同时是多个接口类型。

如示例代码 3-22 所示，新定义了一个接口 ISoutable，而类 Cow 则同时实现接口 IRunable 和 IShoutable，它们用逗号分开，理论上 Cow 还可以实现其他任意的接口。在方法 MultInterface()中，首先定义了一个 Cow 对象 aCow，并将它转换成 IRunable 对象 run 和 IShoutable 对象 shout 分别使用。

示例代码 3-22

```
interface IShoutable
{
    void Shout( );
}
class Cow : IRunable, IShoutable
{
    //省略部分代码
    //实现 IShoutable.Shout()方法
    public void Shout( )
    {
        System.Console.WriteLine("一头牛正在嚎叫......");
    }
}
class Program
{
    static void Main(string[] args)
    {
        MultInterface( );
    }
    static void MultInterface()
    {
```

```
        //定一个 Cow 对象 aCow
        Cow aCow = new Cow( );
        //将 aCow 作为 IRunable 使用
        IRunable run = aCow;
        run.Speed = 2.6f;
        run.Run( );
        //将 aCow 作为 Ishoutable 使用
        IShoutable shout = aCow;
        shout.Shout( );
        //aCow 本身也同时访问 unable 和 IShoutable 的成员
        aCow.Speed = 9.5f;
        aCow.Run( );
        aCow.Shout( );
    }
}
```

从上面的代码可以看出，类 Cow 同时具备了 IRunable 和 IShoutable 的功能，这样就使得类 Cow 同时具备类两种功能（或身份），这就变相实现了多重继承，并且有效避免了多重继承带来的麻烦。当一个类被强制转换成某个它实现的接口类型时，就只能访问该接口的成员。如上面代码中 run 和 shout 都分别只能访问 IRunable 和 IShoutable 的成员。而 aCow 本身则可以同时访问 IRunable 和 IShoutable 的成员。

3.3.4　比较接口和抽象类

接口和抽象类的目的都是定义出最统一最基本的接口，让它们派生出来的所有子类都具有同样的特性，但是它们也有很多不同之处。主要如下：

- 抽象类可以提供字段，它所定义的成员可以有多种可访问性，而接口只能定义公开（public）的成员，而且不能定义字段。
- 抽象类可以为方法提供公有（或默认）的实现，这样子类就可以减少工作量。但是接口定义的方法不能有任何实现，所有的实现都需要子类实现。
- 抽象类的成员不一定需要子类重载，只有抽象成员才一定要重载。而接口的所有成员都必须被子类重载。
- 一个子类只能从一个类继承，但是一个子类可以实现多个接口。

所以，通常如果只是需要定义一系列方法集合，而不提供任何实现，而且这些集合都是可以对外公开时，采用接口。其他情况下，最好是使用类或抽象类。

3.4　异 常 处 理

C#提供了强大的异常处理模块，可以执行自定义异常、分类捕获异常、抛出异常等操作。异常的合理处理，可以大大提高软件的友好性和稳定性。本节将介绍 C#中如何进行异常处理。

3.4.1　用 try…catch 捕获异常

异常和错误的本质区别在于，异常是可预见和可接受的，程序通过对异常的捕获和处

理，可以将异常带来的影响减到最小。而错误通常是程序代码的错误、设计漏洞等，是不可预见的，会给软件带来致命的影响。

C#提供了完善的异常处理机制，通过它可以判断是否有异常发生，可以判断异常的等级分别给出不同的处理，同时还可以抛出自己的异常。另外，.NET 类库提供了常见的异常类，同时又支持自定义异常，使得异常处理更加灵活。

C#中，通过 try…catch 语句块来实现异常的检测和捕获，其中 try 语句将需要检测异常的代码包含起来，catch 语句指定要捕获的异常类型并给出异常发生时的处理代码。如果没有异常发生，则 catch 语句块代码不会执行到。如果异常发生则直接转到 catch 语句的异常处理代码执行，如果该异常没有被捕获，则会一直往上抛出，直到被捕获或被 Windows 捕获。

在示例代码 3-23 的 DevideFunc()函数中，进行除法运算时，如果除数 div 为 0，则会产生异常。因此通过 try 语句将它包含起来作为被检测代码，接着用 catch 语句块来捕获类型为 Exception 的异常，如果捕获到异常，则将异常的消息（Message）打印出来。

示例代码 3-23

```
static void Main(string[] args)
{
    DevideFunc(1, 10);
    DevideFunc(2, 0);
}
static void DevideFunc(int no, int div)
{
    System.Console.WriteLine("[{0}]:Before try statement...", no);
    try
    {
        System.Console.WriteLine("[{0}]:Before DIVDED statement...", no);
        int res;
        res = 20 / div;
        System.Console.WriteLine("[{0}]:After DIVDED statement...", no);
    }
    catch (Exception ex)
    {
        System.Console.WriteLine("[{0}]:Catch an Exception--{1}", no,
        ex.Message);
    }
    System.Console.WriteLine("[{0}]:After try statement...", no);
s}
```

示例代码 3-23 的输出如下，调用 DevideFunc(1, 10)时，由于传入的除数为 10，不会发生异常，则代码会正常运行。调用 DevideFunc(2, 0)时，由于传入除数为 0，语句“res=20/div”会抛出异常，该语句之后的代码段不会执行，而异常捕获语句块内的代码被执行，处理完成后，try…catch 语句后的代码将继续得到执行。

```
[1]:Before try statement...
[1]:Before DIVDED statement...
[1]:After DIVDED statement...
[1]:After try statement...
[2]:Before try statement...
[2]:Before DIVDED statement...
[2]:Catch an Exception--试图除以零。
[2]:After try statement...
```

3.4.2　用 throw 抛出异常

在 C#中，可以通过 throw 语句抛出一个新的异常，或已有异常对象。示例代码 3-24 中，GetValue()函数就根据传入 index 变量的值通过 throw 语句抛出不同的异常。在 Main()函数中，通过 3 次调用 GetValue()函数，并捕获和打印出异常信息，来演示 throw 抛出异常的效果。

示例代码 3-24

```
static void Main(string[] args)
{
    int val;
    try
    {
        val = GetValue(-1);
        System.Console.WriteLine("1—val:{0}", val);
    }
    catch (Exception ex1)
    {
        System.Console.WriteLine("ex1: {0}", ex1.Message);
    }
    try
    {
        val = GetValue(10);
        System.Console.WriteLine("2—val:{0}", val);
    }
    catch (Exception ex2)
    {
        System.Console.WriteLine("ex2: {0}", ex2.Message);
    }
    try
    {
        val = GetValue(9);
        System.Console.WriteLine("3—val:{0}", val);
    }
    catch (Exception ex3)
    {
        System.Console.WriteLine("ex3: {0}", ex3.Message);
    }
}
static int GetValue(int index)
{
    int[] ary = {0,1,2,3,4,5,6,7,8,9};
    if(index < 0)
        throw new Exception("index < 0");
    if(index > 9)
        throw new Exception("index > 9");
    return ary[index];
}
```

示例代码 3-24 的输出如下，s 调用 GetValue(-1)时 index<0，抛出异常。调用 GetValue(10)时 index>9，也抛出异常，两次都不会得到 val 的值，但是异常的消息不一样。调用 GetValue(9)时 index 合法，不会抛出异常，获得并打印出 val 的值。

```
ex1: index < 0
```

```
ex2: index > 9
3—val:9
```

3.4.3　从 Exception 类派生自定义异常

在 C#中，为了提供统一的异常处理接口，所有的异常类都必须从 System.Exception 类继承，该类提供了任何异常类都包含和支持的属性和方法，关键还在于它封装了异常发生位置、错误堆栈等信息的自动生成。表 3-1 中列出了 Exception 类的主要成员。

表 3-1　Exception类主要成员

名　　称	类　　型	说　　明
Exception	构造函数	构造一个异常类，指定其异常消息，发生位置等
Message	只读属性	获取当前异常提供的消息，该消息在构造异常时指定
Source	读写属性	获取和设置引起该异常的应用程序或对象的名称
TargetSite	只读属性	获取引发该异常的方法
ToString	公开方法	创建该异常的字符串表示形式，包括异常发生的位置、名称等信息

在 C#中除了系统提供的异常类型外，开发人员还可以通过自定义异常用来提供更多的异常信息，自定义异常必须直接或间接从 System.Exception 类继承。

如示例代码 3-25 中根据需要从 System.Exception 类派生一个自定义的异常类——MyExBase，同时又从 MyExBase 派生两个异常类 MyDevBase1 和 MyDevBase2。这样 MyExBase、MyDevBase1、MyDevBase2 这 3 个都是自定义异常类，可以通过 catch 语句来捕获这些异常。

示例代码 3-25

```
class MyExBase : Exception
{
   private string  AddtionInfo;
   public string AddtionInfo
   {
      get
      {
         return this. AddtionInfo;
      }
      set
      {
         this. AddtionInfo = value;
      }
   }
}
class MyDevBase1 : MyExBase
{ }
class MyDevBase2 : MyExBase
{ }
```

3.4.4　用多个 catch 子句分级捕获异常

在 C#中，同一个 try 可以配套多个 catch 语句捕获多种类型的异常，根据不同的异常

给出不同的处理。抛出的异常会依次被多个 catch 语句进行类型匹配，直到匹配成功为止。值得注意的是所有子类都是属于父类类型的，所以在多个并列的 catch 语句中，具有继承关系的异常类型，必须按照先特殊后普通（先子类，后父类）的顺序捕获。

示例代码 3-26 中，函数 ThrowAnException()根据传入参数不同引发不同的异常，通过 3 个并列的 catch 语句来分类型捕获异常。首先捕获 MyDevBase1 类型，然后捕获 MyDevBase2 类型，最后捕获 MyExBase 类型，这样就可以对异常进行分类捕获。

示例代码 3-26

```
static void Main(string[] args)
{
    try
    {
        ThrowAnException(1);                //抛出异常
        ThrowAnException(2);
        ThrowAnException(0);
        ThrowAnException(3);
    }
    catch (Exception ex)
    {
        System.Console.WriteLine("Catch a Exception exception.");
    }
}
static void ThrowAnException(int ty)
{
    try
    {
        switch (ty)                         //捕获异常参数
        {
            case 0:
                throw new MyExBase();
            case 1:
                throw new MyDevBase1();
            case 2:
                throw new MyDevBase2();
            default:
                throw new Exception();
                break;
        }
    }
    catch (MyDevBase1 devEx1)
    {
        System.Console.WriteLine("Catch a MyDevBase1 exception.");
    }
    catch (MyDevBase2 devEx2)
    {
        System.Console.WriteLine("Catch a MyDevBase2 exception.");
    }
    catch (MyExBase myEx)
    {
        System.Console.WriteLine("Catch a MyExBase exception.");
    }
 }
```

示例代码 3-26 的输出如下，其中 ThrowAnException(1)引发异常类型 MyDevBase1，被第 1 个 catch 语句捕获。ThrowAnException(2)引发异常类型 MyDevBase2，被第 2 个 catch

语句捕获。ThrowAnException(0)引发异常类型 MyExBase，被第 3 个 catch 语句捕获。ThrowAnException(3)引发 Exception 异常，3 个 catch 语句都不能匹配成功，则继续抛出到 Main()函数中，被 catch(Exception)语句捕获。

```
Catch a MyDevBase1 exception.
Catch a MyDevBase2 exception.
Catch a MyExBase exception.
Catch a Exception exception.
```

3.5　小　结

面向对象思想是任何一门高级开发语言必须具备的方法论，它提供一种高效、可重用、可扩展的解决问题的思考方法。C#作为一门流行的高级开发语言之一，通过类（Class）和接口（Interface）全面灵活简单地支持了面向对象思想。

本章详细介绍了 C#中如何实现类、接口等基本的面向对象知识。通过本章的学习，读者应该掌握以下几个知识点：

- 什么是面向对象？
- 什么是类？什么是对象？
- 在 C#中如何定义类？类包括那些成员？
- 在 C#中类成员具有哪些可访问性？
- 在 C#中如何实现类的继承？
- 在 C#中如何实现方法重载、覆盖等？
- 什么是抽象类和静态类？它们有什么区别？
- 什么是密封类？如何实现它？
- Object 类有什么作用？它有哪些主要成员？
- as 和 is 关键字的功能和区别？
- 在 C#中如何定义和实现接口？
- 在 C#中如何实现多个接口？它们之间的类型关系如何？
- 在 C#中如何实现异常处理？如何抛出异常和处理异常？
- 在 C#中分级捕获异常如何实现？捕获的顺序如何？

第 4 章　C#高级特性

前面两章介绍了 C#最基本的语法特征，本章进一点介绍 C#的一些高级特性，包括委托和事件、泛型实现、扩展方法、分部类等。合理使用这些特性可以让 C#的开发更加快速和高效，同时还可以实现各种不同的设计模式。

4.1　使 用 委 托

委托是一种在 C#中实现函数动态调用的方式，通过委托可以将一些相同类型的函数串联起来依次执行。委托同时还是函数回调和事件机制的基础，本节将详细介绍委托和事件的使用。

4.1.1　按照函数类型定义委托

对于有 C/C++经验的读者来说，函数指针应该并不陌生，函数指针就是指向某种类型函数的指针，通过这种技术可以轻松实现函数的动态调用。在 C#中一切都是对象，函数指针已经不存在，但是可以通过委托（Delegate）实现这种非常灵活有用的机制。

在 C#中，和普通的数据类型一样，函数也具有类型。函数类型是根据函数的返回值和参数来区别的，如果两个函数的返回值类型、参数顺序和类型、参数数量完全相同，那么这两个函数就是同一个类型。例如，下面的几个函数中，FuncA 和 FuncB 就属于同一个类型 int(int,float)，而 FuncC 和 FuncA 由于参数顺序不同，所以不是同一个类型，FuncD 和 FuncC 由于返回值类型不同，所以也不是同一个类型。

```
int FuncA(int var1, float var2);       //int (int, float)
int FuncB(int var1, float var2);       //int (int, float)
int FuncC(float var1, int var2);       //int (float, int)
void FuncD(float var1, int var2);      //void (float, int)
```

在 C#中，委托是一种特殊的数据类型，它表示某种特定类型的函数，并且可以表示多个函数，将这些函数串联起来。使用委托就好像调用函数一样，由于它串联多个函数，所以它可以同时调用多个函数，委托也是事件的基础。

在使用委托之前，首先需要定义委托类型，即定义某种函数类型。在 C#中，通过关键字 delegate 来定义委托，格式如下：

```
delegate returnType DelegateName(Type1 para1, Type2 para2…TypeN paraN);
```

其中，delegate 是关键字，returnType 是委托的返回类型，如果没有则是 void。DelegateName 是委托类型的名称，应该符合 C#中参数的命名规则。Type1 到 TypeN 依次是 N 个参数类型，para1 到 paraN 依次是 N 个参数的名称，如果没有参数，则为空。从 delegate

的格式可以看出，它的定义其实和函数的定义非常相似，只是函数名变成了委托类型名，而且多了 delegate 关键字而已。

如示例代码 4-1 所示，共定义了 3 个委托类型：

- 第一个 DsimpleVoidFunc 是一个没有参数也没有返回值的委托类型。
- 第二个 DParaVoidFunc 没有返回值，但是依次具有 int 和 float 两个参数的委托类型。
- 第三个 DFloatFunc 是具有 float 返回值，同时依次具有 int 和 float 两个参数的委托类型。

示例代码 4-1

```
//定义一个委托类型，它包括 0 个参数，返回类型为 void
delegate void DSimpleVoidFunc();
//定义一个委托类型，它包括两个参数，依次为 int 和 float，返回类型为 void
delegate void DParaVoidFunc(int val1, float val2);
//定义一个委托类型，它包括两个参数，依次为 int 和 float，返回类型为 float
delegate float DFloatFunc(int val1, float val2);
```

注意： 委托实质上是一个类，编译器会根据关键字 delegate 自动生成一个从 System.Delegate 类派生的类。所以，它可以具有可访问性，public、private 等，也包括几个默认的成员函数和属性。读者可以通过 IL 代码看出编译器为委托生成的具体类名称和代码。

4.1.2　用委托动态调用函数

在定义委托类型之后，就可以使用这些委托类型，委托类型的使用和使用其他数据类型一样，首先需要定义一个该类型的变量，然后为变量赋值，最后使用该变量。如下面的代码所示，定义一个委托 DsimpleVoidFunc 类型的变量 voidF。

```
DSimpleVoidFunc voidF;      //定义 DSimpleVoidFunc 委托类型变量 voidF
```

在 C#中，委托变量可以看成是一个特定类型的函数链表。可以通过赋值符号“=”为委托变量指定唯一的函数，也可以用“+=”将新的函数添加到委托所指定的函数链中，也可以通过“–=”从函数链表中删除指定函数。委托变量的使用可以像使用函数那样直接调用，如“voidF()”就是调用 voidF 所指定的函数，也可以通过 Invoke()成员函数来调用。

如示例代码 4-2 所示，首先通过“=”为 voidF 指定唯一的函数 PrintHaHa()为委托链中的函数，然后通过 voidF()调用该函数 PrintHaHa()。然后通过“+=”操作符将 PrintHeHe()添加到委托链中，然后通过 voidF()来依次调用委托链中的函数，此时为 PrintHaHa()和PrintHeHe()。然后通过“–=”操作符从委托链中将 PrintHeHe()移除，再次通过 Invoke()成员函数调用委托链中的函数。

示例代码 4-2

```
static void Main(string[] args)
{
    DSimpleVoidFunc voidF;   //定义 DSimpleVoidFunc 委托类型变量 voidF
    voidF = PrintHaHa;       //为 voidF 赋值 PrintHaHa 函数
```

```
    voidF( );                   //依次调用委托链中的函数，PrintHaHa
    voidF += PrintHeHe;         //将 PrintHeHe 添加到委托链
    voidF( );                   //依次调用委托链中的函数，PrintHaHa->PrintHeHe
    voidF -= PrintHeHe;         //将 PrintHeHe 从委托链中移除
    voidF.Invoke( );            //依次调用委托链中的函数，PrintHaHa
}
static void PrintHaHa()
{
    System.Console.WriteLine("HaHa......");
}
static void PrintHeHe( )
{
    System.Console.WriteLine("HeHe......");
}
```

示例代码 4-2 的输出如下，其中第 1 行 HaHa 是由第 1 次调用 voidF 打印的，第 2～3 行是第 2 次调用 voidF 打印的，第 4 行 HeHe 是第 3 次调用 voidF 打印的。从中可以看出，函数 PrintHeHe()被添加和删除的操作。

```
HaHa......
HaHa......
HeHe......
HaHa......
```

注意：在委托链中的 N 个函数是根据添加的顺序依次从先到后依次调用的，如示例代码 3-2 中是 PrintHaHa()先调用，PrintHeHe()后调用，正好符合添加的顺序。

4.1.3　用委托传递函数参数

在示例代码 4-2 中的委托是不带参数的函数类型，带参数的委托在赋值时不需要指定参数，但是在调用委托链上的函数时需要传入函数参数，这样委托链上所有的函数都会使用这些参数，传入参数在类型和顺序上与直接调用函数一样。如示例代码 4-3 中，floatFunc 的委托链上包含 Add()、Sub()都会使用同样的参数：第 1 个 int 类型的 5，第 2 个 float 类型的 2.0f。

委托所定义的函数类型也可以具有返回值，但是委托链上有多个函数的时候，通过委托变量调用这些函数，只有最后一个被调用的函数会作为委托变量的返回值。如示例代码 4-3 中，floatFunc 第 1 次调用时，委托链上只有 Add()，所以返回值是 Add()的返回值。第 2 次调用 floatFunc 时，由于委托链上最后一个函数是 Sub()，所以返回值是 Sub()的返回值。

示例代码 4-3

```
static void UseParaDelegate()
{
    DFloatFunc floatFunc;                    //DFloatFunc 委托变量
    float result;
    floatFunc = Add;                         //添加第一个函数 Add
    result = floatFunc(5, 2.0f);             //执行函数
    System.Console.WriteLine("1. Result = {0}", result);
    floatFunc += Sub;                        //添加第二个函数 Sub
    result = floatFunc(5, 2.0f);             //执行函数
    System.Console.WriteLine("2. Result = {0}", result);
```

```
    floatFunc += Mult;              //添加第 3 个函数 Mult
    result = floatFunc(5, 2.0f);    //执行函数
    System.Console.WriteLine("3. Result = {0}", result);
    floatFunc -= Mult;              //移除函数 Mult
    floatFunc += Add;               //再一次添加函数 Add
    result = floatFunc(5, 2.0f);    //执行函数
    System.Console.WriteLine("4. Result = {0}", result);
}
//执行加运算
static float Add(int val1, float val2)
{
    System.Console.WriteLine("{0} + {1} = {2}", val1, val2, val1 + val2);
    return val1 + val2;
}
//执行减运算
static float Sub(int val1, float val2)
{
    System.Console.WriteLine("{0} - {1} = {2}", val1, val2, val1 - val2);
    return val1 - val2;
}
//执行乘运算
static float Mult(int val1, float val2)
{
    System.Console.WriteLine("{0} * {1} = {2}", val1, val2, val1 * val2);
    return val1 * val2;
}
```

示例代码 4-3 的输出如下，其中，第 1 次调用 floatFunc 委托链上的函数时，只有 Add()被调用，返回值也是 Add()的返回值。第 2 次调用 floatFunc 委托链上的函数时，依次调用 Add()和 Sub()函数，返回值为最后一个函数的返回值，即 Sub()的返回值。

```
5 + 2 = 7
1. Result = 7
5 + 2 = 7
5 - 2 = 3
2. Result = 3
5 + 2 = 7
5 - 2 = 3
5 * 2 = 10
3. Result = 10
5 + 2 = 7
5 - 2 = 3
5 + 2 = 7
4. Result = 7
```

注意： 委托链上可以具有相同的函数，只要通过“+=”操作添加到委托链上即可。如示例代码 4-3 中第 4 次调用 floatFunc 时，就存在两个 Add()函数。另外，由于委托并不能返回委托链上所有函数的返回值，所以委托函数通常是不需要返回值，或者说委托变量的返回值没有多少实际意义。

4.2　使 用 事 件

事件建立在委托机制之上，通过该机制，某个类在发生某些特定的事情之后，通知其

他类或对象正在发生的事情。事件机制在实际开发中非常实用。本节将介绍事件机制的使用。

4.2.1　定义和发布事件

在 C#中，当一个类在发生某个定义的事情（或事件）之后，该类可以通过事件机制通知已经注册的类或对象正在发生的事情。从本质上来说，事件实际就是委托，但是它通常是特定的函数类型，具有以下特点：

- 事件发行者（类）确定何时引发事件，事件订阅者确定如何响应该事件。
- 一个事件可以有多个订阅者。一个订阅者可以处理来自多个发行者的多个事件。
- 没有订阅者的事件永远不会被调用。
- 如果一个事件有多个订户，当引发该事件时，会同步调用多个事件处理程序。
- 在.NET Framework 类库中，事件是基于 EventHandler 委托和 EventArgs 基类的。

在 C#中，通过 event 关键字定义事件，格式如下，accessAttr 是事件的可访问属性，可以是 private、public、protected，但是由于事件可以被其他类或对象知道，所以通常是 public。event 是关键字，表示该成员是该类的事件。DelegateType 是该事件的响应函数类型，必须是可访问的委托类型。eventName 是该事件的名称，命名规则和类成员的命名规则相同。

```
格式：accessAttr event DelegateType eventName;
例子：public event PriceChangedEventHander PriceChanged;
```

事件响应函数委托通常是没有返回值，有 sender 和 arg 两个参数，前者是 object 类型，表示事件的发起者。后者是事件参数，通常是从 System.EventArgs 类派生得来。所以在定义一个事件之前，要先定义事件的参数类型，该类型包含了事件发起者需要提供给事件订阅者的信息。

另外，引发事件实际上就是调用委托变量，但是在事件定义之后，该变量默认为 null，直接引发会产生异常，所以调用之前要判断事件是否为 null（即是否已经被订阅）。通常通过定义 OnXX()的函数来引发 XX 事件，在该函数中首先判断事件是否被订阅，如果被订阅则引发事件。

如示例代码 4-4 所示，Shop 类是一个商店类，当商店的商品价格发生变化时，通常需要通知到现有的客户，便于客户选购商品，在这里就可以使用事件机制来实现。首先，定义价格变化事件的参数类——PriceChangedEventArgs，它从 EventArgs 类派生而来，包括商品的名称（Name）和新价格（Price）两个信息。然后，定义该事件的处理委托类型——PriceChangedEventHander，它没有返回值，第一个参数为 object 类型，表示引发该事件的类型，第二个参数为 PriceChangedEventArgs 类型，包含事件的详细信息。

定义事件参数类和响应委托类型之后，就可以在 Shop 类中定义具体的事件——PriceChanged，它实际上就是 PriceChangedEventHander 类型委托。最后，在 OnPriceChanged() 中引发 PriceChanged 事件，将当前类作为事件的发生者，在引发事件之前先判断是否已经注册（即是否为 null）。

示例代码 4-4

```
/// <summary>
/// 商品价格变化时发生的事件参数
/// </summary>
public class PriceChangedEventArgs : EventArgs
{
    //构造函数，初始化商品名称和新价格
    public PriceChangedEventArgs(string nm, float pr)
    {
        this. Name = nm;
        this. Price = pr;
    }
    //表示价格变化的商品的名称
    private string _Name;
    public string Name
    {
        get
        {
            return this._Name;
        }
    }
    //表示价格变化之后的商品的价格
    private float _Price;
    public float Price
    {
        get
        {
            return this._Price;
        }
    }
}
//定义价格变化事件的委托
public delegate void PriceChangedEventHander(object sender, PriceChanged-
EventArgs arg);
/// <summary>
/// 商店类，价格变化事件的发生类
/// </summary>
public class Shop
{
    //通过 event 关键字定义价格变化事件
    public event PriceChangedEventHander PriceChanged;
    //引发事件函数
    protected void OnPriceChanged(PriceChangedEventArgs arg)
    {
        //如果事件已经注册，则通过委托调用函数的方式通知事件订阅用户
        if(this.PriceChanged != null)
        {
            this.PriceChanged(this, arg);
        }
    }
    //更新商品的名称，并引发商品价格变化事件
    public void UpdatePrice(string nm, float newPrice)
    {
        //创建 PriceChanged 事件参数
        PriceChangedEventArgs arg = new PriceChangedEventArgs(nm, newPrice);
        //引发 PriceChanged 事件
        this.OnPriceChanged(arg);
    }
```

```
    public Shop(string nm)
    {
        this. Name = nm;
    }
    //商店的名称
    private string _Name;
    public string Name
    {
        get
        {
            return this._Name;
        }
    }
}
```

事件的引发者 sender 参数通常用来获取引发者的实际类型和更多详细信息。示例代码 4-4 中只是给出了事件的通常格式，而实际上事件响应函数可以是任何的委托类型，即任意数量和类型的参数、任意类型的返回值。

技巧：一套合理的事件命名规则会让代码可读性更好，作者比较常用的格式是：定义事件 XXXX，事件参数类为 XXXXEventArgs，事件处理委托类型为 XXXXEventHander，引发事件的方法为 OnXXXX()。

4.2.2　订阅和处理事件

前面通过 Shop 类介绍了事件的定义和引发，本节将通过一个 Customer（客户）类来演示如何订阅和处理事件。订阅和处理事件的一个核心元素是事件响应（处理）函数，事件响应函数是符合要订阅的事件委托类型的函数，它通常根据事件的引发者和参数进行相应的处理。

由于事件的本质是委托，所以事件的订阅实际上就是通过“+=”运算符将当前类的事件响应函数添加到事件的委托链中，在引发事件时就可以调用该处理函数。相反，取消订阅则是通过“–=”操作将当前类的事件响应函数从事件委托链中移除，这样在引发事件时就不会调用该函数。

示例代码 4-5 定义 Customer 类，其中 Shop_PriceChanged()方法就是 PriceChanged 事件响应函数，在该函数中将 sender 转换成 Shop 类型，并打印出事件具体信息。在实际项目开发中，事件响应函数会更加复杂。

示例代码 4-5

```
/// <summary>
/// 客户类，用于订阅 PriceChanged 事件
/// </summary>
public class Customer
{
    public Customer(string nm)
    {
        this. Name = nm;
    }
    //客户姓名
    private string _Name;
```

```
    public string Name
    {
        get
        {
            return this._Name;
        }
    }
    //PriceChanged 事件响应函数
    public void Shop_PriceChanged(object sender, PriceChangedEventArgs arg)
    {
        //将 sender 转换成 Shop 类型变量，并打印提示信息
        Shop sp = sender as Shop;
        if (sp != null)
            System.Console.WriteLine("{0}收到{1}: {2}新价格为{3}￥",
                            this._Name, sp.Name, arg.Name, arg.Price);
    }
}
```

示例代码 4-6 演示如何通过“+=”和“–=”运算符订阅和取消订阅 PriceChanged 事件，首先定义 Shop 对象 shop1 和 shop2，用来引发事件，定义 Customer 对象 cust1 和 cust2，用来订阅和取消订阅事件。然后通过“+=”运算符实现 cust1 和 cust2 订阅 shop1 和 shop2 的 PriceChanged 事件，然后通过“–=”运算符取消订阅。

示例代码 4-6

```
static void Main(string[] args)
{
    //定义两个 Shop 对象，用来引发事件
    Shop shop1 = new Shop("Shop1");
    Shop shop2 = new Shop("Shop2");
    //定义两个 Customer 对象，用来订阅并处理事件
    Customer cust1 = new Customer("Customer1");
    Customer cust2 = new Customer("Customer2");
    //cust1 订阅 shop1 的 PriceChanged 事件
    shop1.PriceChanged += cust1.Shop_PriceChanged;
    //shop1 更新商品价格，引发 PriceChanged 事件
    System.Console.WriteLine("1.shop1.UpdatePrice(\"Goods1\", 2.2f)......");
    shop1.UpdatePrice("Goods1", 2.2f);
    //cust2 订阅 shop2 的 PriceChanged 事件
    shop2.PriceChanged += cust2.Shop_PriceChanged;
    //shop2 更新商品价格，引发 PriceChanged 事件
    System.Console.WriteLine("2.shop2.UpdatePrice(\"Goods2\", 3.4f)......");
    shop2.UpdatePrice("Goods2", 3.4f);
    //cust2 订阅 shop1 的 PriceChanged 事件
    shop1.PriceChanged += cust2.Shop_PriceChanged;
    //shop1 更新商品价格，引发 PriceChanged 事件
    System.Console.WriteLine("3.shop1.UpdatePrice(\"Goods3\", 55.2f)......");
    shop1.UpdatePrice("Goods3", 55.2f);
    //cust1 取消订阅 shop1 的 PriceChanged 事件
    shop1.PriceChanged -= cust1.Shop_PriceChanged;
    //shop1 更新商品价格，引发 PriceChanged 事件
    System.Console.WriteLine("4.shop1.UpdatePrice(\"Goods4\", 9.2f)......");
    shop1.UpdatePrice("Goods4", 9.2f);
}
```

示例代码 4-6 的输出如下所示，从中可以看出，一个对象的事件可以被多个对象订阅，

一个对象也可以订阅多个对象的事件。

```
1.shop1.UpdatePrice("Goods1", 2.2f)......
Customer1 收到 Shop1: Goods1 新价格为 2.2￥
2.shop2.UpdatePrice("Goods2", 3.4f)......
Customer2 收到 Shop2: Goods2 新价格为 3.4￥
3.shop1.UpdatePrice("Goods3", 55.2f)......
Customer1 收到 Shop1: Goods3 新价格为 55.2￥
Customer2 收到 Shop1: Goods3 新价格为 55.2￥
4.shop1.UpdatePrice("Goods4", 9.2f)......
Customer2 收到 Shop1: Goods4 新价格为 9.2￥
```

4.3　使用泛型

泛型将类型参数的概念引入到 C#中，通过泛型可以最大化代码重用，它将代码类型的指定推迟到客户端代码，大大提高了集合类的效率。本节将结合集合类介绍泛型在 C#中的使用。

4.3.1　定义泛型

泛型是在 C#中实现类型参数化的机制，数据类型作为参数用来定义一个新的数据类型。通过这种机制，可以编写出一套接口，该接口基于一个或多个假设的类型，只有在使用这个接口的时候才能确定它的真正类型。

在 C#中，通过尖括号“< >”将类型参数括起来，表示泛型，如下面的代码所示，其中 FXClass 是一个泛型类，<T>表示 T 是一个假设的类型，在 FXClass 中该类可以认为 T 是已知类型。同样，函数 FXFunction 是一个泛型函数，它也认为 T 是已知类型。

```
class FXClass<T>
T FXFunction<T>(T para)
```

在定义了泛型类之后，默认情况下 T 是任何类型都可以，所以可以用实际的数据类型代替 T 来声明某个实际要使用的类型。值得注意的是，对于同一个泛型定义，不同的类型作为参数所产生的新类型是两个不同的新类型。如下所示，intFxCls 和 objFxCls 分别是 FXClass 在 int 和 object 上的两个实例，它们属于不同的类型，即 FXClass<int>和 FXClass<Object>是两个不同的类型。

```
FXClass<int> intFxCls;
FXClass<Object> objFxCls;
```

当泛型需要多个类型参数时，多个参数类型用逗号“，”分隔，但名字不同，而且都在“< >”内，如下面代码所示。前者需要两个类型参数，后者需要三个类型参数。

```
class FX2Class<T, U>
class FX3Class<T, U, Y>
```

多个类型参数在定义类的时候，必须给出相同数量的类型，如下面代码所示。

```
FX2Class<int, float> fx2Cls;
class FX3Class<int, int, float> fx3Cls;
```

另外，如果在定义泛型时，希望被使用的参数类型是实现了特定将接口的类型，那么可以通过 where 关键字和冒号“:”运算符指定参数类型的父类（或接口），所有作为参数的类都必须是这个类（或接口）或它的子类。如下面的代码所示，where 关键字要求 T 必须是实现了 IComaparable 接口的类型。所以，strFxCls 和 intFxCls 都是正确的，string 和 int 都实现了 IComparable 接口，但是，objFxCls 不正确，因为 object 并没有实现 IComparable 接口，而 FxClass<T>在定义的时候限制类型必须为 IComparable 的子类。

```
class FxClass<T> where T:IComparable{}
----------------------------------------
FxClass<string> strFxCls;        //正确
FxClass<int> intFxCls;           //正确
FxClass<object> objFxCls;        //错误，object 没有实现 IComparable 接口
```

C#中的泛型具有以下特点，合理地使用泛型可以让代码变得更加高效和安全。

- 泛型类型可以最大限度地重用代码、保护类型的安全。
- 通过减少拆箱和装箱操作，从而大大提高性能。
- 可以泛型化的 C#语言元素很多，包括泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 可以通过冒号“:”运算符对泛型类进行约束以访问特定数据类型的方法。

泛型最常用的地方就是用于定义集合类，在.NET 类库中，几乎所有集合类都具有泛型版本，由于泛型版本性能更高，所以强烈建议使用泛型版本的集合类。在 4.4 节将介绍常见的集合类。

4.3.2　泛型实例

通常，在需要对多种类型进行操作，而且不知道有哪些类型需要操作，也不明确将来会有多少类型需要支持这样的操作，这时就可以考虑泛型。如果要操作的类型具有相同的父类或是都实现了某个接口，那么可以通过 where 和“:”对数据泛型的类型参数进行约束。

在示例代码 4-7 中，DisaplyShell<T>是一个泛型类，它试图打印出任何类型的对象的字符串格式，所以采用泛型，它的字段_Value 的数据类型为 T，即和泛型的类型参数一致。DisplayShell<int>的_Value 字段数据类型为 int，构造函数的参数 at 类型也为 int。同理，DisplayShell<string>的_Value 字段数据类型为 string，构造函数的参数 at 类型也为 string。

泛型函数 Display<U>(U val)则是显示特定类型的对象，它的参数 val 的数据类型取决于类型参数 U。如果类型参数 U 为 int，那么 val 的类型为 int，该函数变成：Display<int>(int val)。如果 U 为 string，那么 val 的类型为 string，该函数变成：Display<string>(string val)。

示例代码 4-7

```
//泛型类 DisplayShell 为所有类提供一个外壳，用来显示这些类，T 为类型参数
class DisplayShell<T>
{
   //表示当前对象所封装的实际对象，它的类型由参数类型 T 决定
   private T  Value;
   //构造函数，传入该类的初始值，at 的类型由参数类型 T 决定
   public DisplayShell(T at)
   {
      this._Value = at;
```

```
    }
    public void DisplayMe()                    //显示当前对象的实际所指向对象
    {
        System.Console.WriteLine("DisplayMe(): {0}", this._Value);
    }
}
class Program
{
    //泛型函数，显示指定的对象，U 为具体的类型参数
    public static void Display<U>(U val)
    {
        System.Console.WriteLine("Display<U>(): {0}", val);
    }
    static void Main(string[] args)
    {
        //创建泛型 DisplayShell 的 int 类型特例对象
        DisplayShell<int> dsInt = new DisplayShell<int>(20);
        dsInt.DisplayMe( );                    //显示整数
        //创建泛型 DisplayShell 的 string 类型特例对象
        DisplayShell<string> dsStr = new DisplayShell<string>("Hi，你好！");
        dsStr.DisplayMe( );                    //显示字符串
        //创建泛型 DisplayShell 的 Program 类型特例对象
        DisplayShell<Program> dsProg = new DisplayShell<Program>(new
        Program( ));
        dsProg.DisplayMe( );                   //显示 Program 对象
        //使用泛型函数 Display<U>的 int 实例
        Display<int>(30);                      //显示 30
        //使用泛型函数 Display<U>的 string 实例
        Display<string>("你也好！");           //显示字符串
    }
}
```

示例代码 4-7 的输出如下，可以看出泛型类 DisplayShell<T>根据实际类型不同，它的字段_Value 的类型和值也不同。同样地，泛型函数 Display<U>也根据实际类型不同，打印出不同的参数值。

```
DisplayMe(): 20
DisplayMe(): Hi，你好！
DisplayMe(): UseFanXing.Program
Display<U>(): 30
Display<U>(): 你也好！
```

4.4　泛型集合类

泛型在.NET 类库中最主要的用途在于实现集合类，.NET 类库提供了多种常见的集合类，本节将介绍两个主要的泛型集合类，List<T>和 Dictionary<TKey,TValue>。

4.4.1　使用泛型列表 List<T>

在.NET 类库中，泛型最主要用在集合类的实现中，.NET 类库提供了很多中集合类，

包括 ArrayList、List、List<T>、LinkedList<T>、HashTable、HashSet、Dictionary<T>等，后面几节将详细介绍泛型实现的几个常用集合类。

列表是简单和常见的集合类之一，在.NET 类库中，泛型类 List<T>实现了可以通过索引访问强类型的列表。泛型类 List<T>的参数类型 T 可以是任何可访问的数据类型，值类型和引用类型均可，可以通过中括号“[ ]”运算符像访问数组那样访问 List<T>的元素，它的索引也是从 0 开始计数。

List<T>类提供一个类似数组的容器，但是与数组的固定大小不同，它会根据列表中元素的个数自动调整列表容量，分配和释放所占用的资源。值得注意的是，List<T>的容量往往是大于实际的元素数量，这样是为了防止每次添加和移除元素时都进行内存分配和释放，从而提高性能。

List<T>提供一系列成员方法，通过这些成员可以访问列表中元素的数量，对列表进行排序、查找等，表 4-1 列出了 List<T>的主要成员。

表 4-1　List<T>主要成员

成 员 名 称	成 员 说 明
Capacity	Public读写属性，获取或设置当前列表的容量，即当前列表当前可以容纳的元素个数
Count	Public只读属性，获取当前列表的元素个数，肯定小于或等于Capacity
Add	Public方法，将元素添加到列表的结尾处，如果该元素已经存在，仍然重复添加
AddRange	Public方法，将指定集合中的所有元素添加到当前列表的结尾处，如果某元素已经存在，仍然重复添加
Insert	Public方法，将元素插入当前列表的指定索引处，索引越界会产生异常
InsertRange	Public方法，将指定集合中所有元素插入到当前列表的指定索引处，索引越界会产生异常
Clear	Public方法，移除当前列表中的所有元素
Remove	Public方法，从当前列表中移除指定元素
RemoveAll	Public方法，从当前列表中移除符合指定条件的所有元素
RemoveAt	Public方法，从当前列表中移除指定索引处的元素，索引越界会产生异常
RemoveRange	Public方法，从当前列表中移除指定索引处开始的N个元素，索引或数量越界会产生异常
Sort	Public方法，对当前按列表全部或部分元素进行排序，可以指定排序所用的比较器（委托类型），如果范围越界会产生异常
Reverse	Public方法，将当前列表全部或部分元素的顺序反转，索引越界会产生异常
BinarySearch	Public方法，使用二分法在已经排序的列表或它的一部分中查找特定元素，使用前必须先保证列表是正确排序的，否则不能保证正确找到，如果范围越界会产生异常
Contains	Public方法，确定某个元素是否在当前列表中
Exists	Public方法，确定当前列表是否包含符合指定条件的元素，条件是一个委托类型
Find	Public方法，搜索并返回当前列表中第一个满足指定条件的元素，条件是一个委托类型
FindAll	Public方法，搜索并返回当前列表中满足指定条件的所有元素，条件是一个委托类型
FindIndex	Public方法，搜索并返回当前列表中第一个满足指定条件的元素索引，条件是一个委托类型，索引从0开始计数

续表

成 员 名 称	成 员 说 明
FindLast	Public方法，搜索并返回当前列表中最后一个满足指定条件的元素，条件是一个委托类型
FindLastIndex	Public方法，搜索并返回当前列表中最后一个满足指定条件的元素索引，条件是一个委托类型，索引从0开始计数
IndexOf	Public方法，返回当前列表全部或部分中第一个满足指定条件的元素索引，索引从0开始计数
LastIndexOf	Public方法，返回当前列表全部或部分中最后一个满足指定条件的元素索引，索引从0开始计数
ForEach	Public方法，对当前列表中的每个元素执行指定操作，操作是一个委托类型
TrueForAll	Public方法，确定是否当前列表中的所有元素都与满足指定的条件，条件是一个委托类型
TrimExcess	Public方法，将列表的容量设置为列表中的实际元素数目
ToArray	Public方法，将当前列表中的元素复制到新数组中，只是引用复制，元素本身并没有产生多份备份

从表 4-1 中可以看出，List<T>提供了丰富的接口访问、查找列表中的元素，在下面几节中将介绍部分常用成员函数的使用。值得注意的是，在这些成员中，如果需要满足某个条件的函数，那么这个条件实际上是通过泛型委托 Predicate<T>提供的，这个委托返回 bool 类型，输入是一个类型为 T 的参数，定义如下：

```
public delegate bool Predicate<T>( T obj );
```

在 List<T>在进行条件判断时，将需要判断的元素（类型为 T）作为该条件委托（类型为 Predicate<T>）的参数输入，如果返回结果为 true，则该元素符合条件，否则不符合条件，具体使用实例见 4.4.4 节。

4.4.2　添加元素到 List<T>

在使用 List<T>之前，首先需要创建一个 List<T>对象，和创建其他类型的对象一样，通过 new 关键字类创建 List<T>对象，但是在创建时要明确指定参数类型 T 的实际类型，创建之后该对象只能包含指定类型（及其子类型）的元素。

如下面的代码所示，intLst 是通过 new 关键字创建的 List<int>类型对象，它只能包含 int 类型的元素，为它添加非 int 类型元素是不合法的。同样地，strLst 只能包含 string 类型的元素，objLst 可以包含任何类型的成员，因为任何类型都是从 object 继承，都属于 object 类型。

```
List<int> intLst = new List<int>( );             //创建 int 类型的列表
List<string> strLst = new List<string>( );       //创建 string 类型的列表
List<object> objLst = new List<object>( );       //创建 object 类型的列表
```

在创建了 List<T>对象之后，可以通过表 3-1 中列出的 Add()、AddRange()、Insert()、InsertRange()这 4 个方法为它添加元素，但是元素类型必须为指定的类型（T 的类型）。这 4 个方法的定义如下：

```
public void Add(T item);
public void AddRange(IEnumerable<T> collection);
public void Insert(int index,T item);
public void InsertRange(int index,IEnumerable<T> collection);
```

其中，T 为列表 List<T>的元素类型，item 是表示要添加或插入的元素，index 表示要插入的元素的起始索引，该索引从 0 开始，并且一定要在当前列表的范围内。collection 是实现了 IEnumerable<T>接口的对象，通过它可以获取一组类型为 T 的元素，即要添加的元素。

实例代码 4-8 演示如何添加和插入元素到 List<T>中，为了简单一些，这里就用 List<int>作为例子。首先创建 int 数组 ary1 和 ary2 备用，它们是实现了 Ienumerable<int>的对象。然后通过 new 创建一个 List<int>对象 intLst，通过 Add()先后添加 10 和 20 到列表末尾，此时 intLst 中的元素为｛10，20｝。然后，通过 Insert()先后在索引为 0 和 1 的位置分别插入 5 和 8，此时 intLst 中的元素为｛5，8，10，20｝。接着，通过 AddRange()方法将 ary1 的所有元素添加到列表末尾，此时 intLst 中的元素为｛5，8，10，20，1，3，7｝。最后，通过 InsertRange()方法将 ary2 的所有元素插入到索引为 2 的位置，此时 intLst 中的元素为｛5，8，2，4，6，10，20，1，3，7｝。

示例代码 4-8

```
static void AddInsertList()
{
    int[] ary1 = new int[] { 1, 3, 7 };//创建两个 int 数组 ary1 和 ary2 备用
    int[] ary2 = new int[] { 2, 4, 6 };
    List<int> intLst = new List<int>( );//创建 int 类型的列表，新建时为空
    intLst.Add(10);                         //添加 10 到末尾，intLst 为{10}
    intLst.Add(20);                         //添加 20 到末尾 intLst 为{10, 20}
    intLst.Insert(0, 5);     //在索引为 0 的位置插入 5，intLst 为{5,10,20}
    intLst.Insert(1, 8);    //在索引为 1 的位置插入 8，intLst 为{5,8,10,20}
    intLst.AddRange(ary1); //将 ary1 添加到末尾，intLst 为{5,8,10,20,1,3,7}
    intLst.InsertRange(2, ary2);
            //在索引为 2 的位置插入 ary2, intLst 为{5,8,2,4,6,8,10,20,1,3,7}
}
```

注意： 当用 Add()和 AddRange()添加元素到列表中时，列表中原有元素的索引不会改变，但是 Insert()和 InsertRange()插入元素之后，列表中原有元素的索引会发生变化。另外，List<T>会根据当前元素的个数（Count）自动分配或释放它所占用的空间，所以 List<T>的容量往往大于它的元素个数。

4.4.3　遍历 List<T>的元素

遍历 List<T>中的元素，最常见的方法是通过 foreach 语句从第 0 个元素开始，依次取出列表中的所有元素，取出元素的类型为 T。如下面的代码所示，由于 intLst 是 List<int>类型，所以从它获取的元素 val 类型也为 int，然后在 foreach 的代码块中就可以使用元素 val，这里只是打印出 val 的值。

```
foreach (int val in intLst)
{
    System.Console.WriteLine(val);
}
```

当然，也可以用 for 语句，结合元素的索引和“[ ]”运算符，遍历所有列表中的所有元素，如下面的代码具有和上面的 foreach 同样的效果。其中，Count 是 List<T>的 public 只读属性，返回当前列表中元素数量。中括号“[ ]”运算符可以访问数组一样访问 List<T>的值，但是索引范围必须是 0 到 Count-1。

```
for (int i = 0; i < intLst.Count; i++)
{
    System.Console.WriteLine(intLst[i]);
}
```

除此之外，在 List<T>中还提供了一个成员方法 ForEach()，通过它可以在遍历列表中所有元素时，对遍历出来的元素执行指定操作，相当于前面代码中的 foreach 和打印 val 值都用一个语句来完成。ForEach()方法的定义如下：

```
public void ForEach(Action<T> action);
public delegate void Action<T>(T obj);
```

其中，ForEach()的参数 action 是类型为泛型委托 Action<T>的一个实例，用来指定对每个元素要执行的操作。而 Action<T>则是一个没有返回值，且只有一个类型为 T 的参数的委托。由此可见，通过 Action<T>将对元素的动作实现转移到调用者，使得扩展性更好。

示例代码 4-9 演示 ForEach()方法的具体使用。其中，void PrintInt(int)和 void PrintLevel(int)都是符合泛型委托 Action<int>的函数类型，分别用于打印具体的分数，打印分数对应的等级。在 UseFroeach()中，先创建并初始化包含 3 个分数的 List<int>对象 scrLst，然后，先通过 ForEach()对 scrLst 中所有元素执行 PrintInt()操作，然后再对 scrLst 中所有元素执行 PrintLevel()操作。

示例代码 4-9

```
static void PrintInt(int val)
{
    System.Console.WriteLine("val = {0}", val);     //打印出数值
}
static void PrintLevel(int val)
{
    if (val < 60)                                   //小于 60 为 D 等
    {
        System.Console.WriteLine("{0} is LEVEL {1}", val, 'D');
    }
    else if (val <= 80)                             //小于 80 为 C 等
    {
        System.Console.WriteLine("{0} is LEVEL {1}", val, 'C');
    }
    else if (val <= 90)                             //小于 90 为 B 等
    {
        System.Console.WriteLine("{0} is LEVEL {1}", val, 'B');
    }
    else if (val <= 100)                            //90 以上为 A 等
    {
```

```
        System.Console.WriteLine("{0} is LEVEL {1}", val, 'A');
    }
}
static void UseForeach( )
{
    int[] scores = { 40, 80, 95};                   //创建分数
    List<int> scrLst = new List<int>( );            //创建 List<int>对象 scrLst
    scrLst.AddRange(scores);                        //添加分数 scores 到 scrLst
    scrLst.ForEach(PrintInt);                       //打印所有分数
    scrLst.ForEach(PrintLevel);                     //打印出所有分数对应的等级
}
```

示例代码 4-9 的输出如下，可见，ForEach()从索引为 0 的元素开始，依次对 scrLst 中的所有元素都执行了指定的操作。

```
val = 40
val = 80
val = 95
40 is LEVEL D
80 is LEVEL C
95 is LEVEL A
```

4.4.4　对 List<T>进行排序

List<T>作为列表，排序也是它的一个基本功能。List<T>可以通过 Sort()对列表中的元素进行从小到大排序，同时 Sort()还接收自定义比较器，这样开发人员可以根据需要指定希望的比较方式。Sort()方法的 3 个常用版本的定义如下：

```
public void Sort();
public void Sort(IComparer<T> comparer);
public void Sort(int index,int count,IComparer<T> comparer);
```

其中，第 1 个版本是通过默认的比较器对列表中所有元素进行从小到大排序，如果类型 T 没有默认比较器，也没有实现接口 IComparer<T>，即 T 不能进行比较，那么会产生异常。参数 comparer 是一个实现了 IComparer<T>接口的类型对象，Sort()通过 comparer 接口的 Compare()对元素进行比较。第 3 个版本是部分元素进行比较，index 表示起始索引（0 开始计数），count 表示要排序的元素个数。

IComparer<T>接口只有一个成员 Compare(T x, T y)，通常情况下，如果 x 小于 y 返回负数，x 等于 y 返回 0，x 大于 y 返回整数。如示例代码 4-10 中的 MyIntComparer 类，该类实现接口 IComparer<int>，成员 Compare(int x, int y)根据 x 和 y 的绝对值 x 和 y 进行比较。

示例代码 4-10 演示 Sort()方法和 IComparer<T>的使用，泛型方法 PrintList<T>()用来打印列表中的元素，该函数在后面的例子中会继续使用。在 UseSort()方法中，ary 是最原始的数据，首先，用默认的比较器对元素按照从小到大排序（负数小于整数）。然后，用自定义整数比较器 MyIntComparer 对象 mic 对列表中的元素按照绝对值排序。最后，用 mic 对列表中第 2 个开始的 3 个元素按照绝对值排序。

示例代码 4-10

```
//自定义整数比较器，按照整数的绝对值进行比较
class MyIntComparer:IComparer<int>
```

```
{
    //重写 int 比较器，|x|>|y|返回正数，|x|=|y|返回 0，|x|<|y|返回负数
    public int Compare(int x, int y)
    {
        int x1 = Math.Abs(x);                    //x 的绝对值 x1
        int y1 = Math.Abs(y);                    //y 的绝对值 y1
        return x1 - y1;
    }
}
//在同一行上打印列表中的元素
static void PrintList<T>(string hint, List<T> lst)
{
    System.Console.Write(hint + ":");
    foreach (T val in lst)                       //遍历并打印列表中的元素
    {
        System.Console.Write("{0} ", val);
    }
    System.Console.WriteLine();
}
static void UseSort()
{
    int[] ary = {9, 8, -11, 10, -3, 2};         //准备数据，添加到列表中
    MyIntComparer mic = new MyIntComparer();
    List<int> intLst = new List<int>( );
    intLst.AddRange(ary);                        //将数据添加到 List<int>中
    PrintList<int>("排序前", intLst);            //打印数据
    intLst.Sort( );                              //使用默认比较器进行排序
    PrintList<int>("默认排序后", intLst);        //打印数据
    intLst.Clear( );                             //重新准备数据
    intLst.AddRange(ary);
    intLst.Sort(mic);                     //用自定义的比较器进行排序，即按绝对值排序
    PrintList<int>("绝对值排序后", intLst);      //打印数据
    intLst.Clear( );                             //重新准备数据
    intLst.AddRange(ary);
    intLst.Sort(2, 3, mic);               //对第 2 个开始的 3 个元素按绝对值排序
    PrintList<int>("部分排序后", intLst);        //打印数据
}
```

示例代码 4-10 的输出如下，默认排序之后元素按照从小到大排序，负数小于 0 和整数。按绝对排序之后，–3 大于 2，可见 MyIntComparer 起作用了。同样部分排序的时候，第 0 和 1 两个元素和最后一个元素都没有参与排序。

```
排序前:9 8 -11 10 -3 2
默认排序后:-11 -3 2 8 9 10
绝对值排序后:2 -3 8 9 10 -11
部分排序后:9 8 -3 10 -11 2
```

技巧： 如果希望对元素从大到小排序，可以先用 Sort()方法对元素从小到大排序，然后再用 Reverse()将列表中的元素倒置，就变成从大到小排序。

4.4.5　在 List<T>中查找元素

List<T>提供多个成员函数在列表中查找元素，通过这些方法返回找到的一个或多个元

素，或返回找到元素的索引。这些成员包括 Find()、FindAll()、FindIndex()、FindLast()、FindLastIndex()、IndexOf()、LastIndexOf()，这些成员的查找条件可以是简单的元素相同，也可以通过 Predicate<T>委托类型指定自定义的查询条件，本节介绍几个典型的成员。

通过 Find()、FindLast()和 FindAll()可以查找类表中符合指定条件的第一个或多个元素。它们的定义如下，Find()返回第一个符合指定条件的元素，FindLast()返回列表中最后一个符合条件的元素，FindAll()则返回列表中符合指定条件的所有元素，并通过一个新的 List<T>列表返回。

```
public T Find(Predicate<T> match);
public T FindLast(Predicate<T> match);
public List<T> FindAll(Predicate<T> match);
```

其中，match 是一个泛型委托 Predicate<T>，它接受一个 T 类型参数，返回 bool 值，如果满足条件返回 true，否则返回 false。Find()等方法依次将列表中的元素（类型为 T）传入到该委托 match 中，判断是否符合指定条件。

由于判断条件是一个委托类型，所以开发人员可以根据需要指定各种复杂的判决条件。如示例代码 4-11 所示，函数 bool BeginWithA(string)判断给定字符串是否以字母 A 开始，如果是则返回 true，否则返回 false，BeginWithA()也是属于 Predicate<string>委托类型的实例。UseFinds()方法首先创建一个 List<string>对象 strLst，然后先后通过 Find()、FindLast()、FindAll()找出列表中的第一个、最后一个和所有的以字母 A 为开始的字符串，并打印出来。

示例代码 4-11

```
//泛型 Predicate<string>，判断一个字符串是否以字母 A 开始
static bool BeginWithA(string str)
{
    if (string.IsNullOrEmpty(str))       //如果字符串为空，则返回 false
        return false;
    char ch = str[0];                    //取得第一个字符
    return ch == 'A';                    //如果等于 A 返回 true，否则返回 false
}
static void UseFinds( )
{    //准备数据，添加到列表中
    string[] ary = { "begin", "ABC", "abc", null, "Aaha", "hehe", "Aig",
    "end"};
    MyIntComparer mic = new MyIntComparer( );
    List<string> strLst = new List<string>( );
    strLst.AddRange(ary);                //将数据添加到 List<string>中
    PrintList<string>("原数据", strLst);          //打印数据
    string str1 = strLst.Find(BeginWithA);        //查找第一个 A 开始的字符串
    System.Console.WriteLine("第一个字符串:{0}", str1);
    string str2 = strLst.FindLast(BeginWithA); //查找最后一个 A 开始的字符串
    System.Console.WriteLine("第一个字符串:{0}", str2);
    List<string> allStr = strLst.FindAll(BeginWithA);
                                                  //查找所有 A 开始的字符串
    PrintList<string>("全部字符串", allStr);
}
```

示例代码 4-11 的输出如下，从中可以看出，所有 Find()的确找出列表中第一个以字母

A 开始的字符串 ABC，FindLast()也找出列表中最后一个以字母 A 开始的字符串 Aig，FindAll()则找出了列表中所有以字母 A 开始的字符串。

```
原数据:begin ABC abc  Aaha hehe Aig end
第一个字符串:ABC
第一个字符串:Aig
全部字符串:ABC Aaha Aig
```

由于篇幅关系，本节只是介绍了 Find 系列函数的几个常用版本，关于其他几个版本，读者可以查看 MSDN 或是微软官方论坛。

4.4.6　移除 List<T>的元素

List<T>还提供几个方法用来从列表中移除元素，这些方法同样可以移除指定的元素，也可以移除一个或多个满足指定条件的元素，从表 4-1 中可以看出，它们包括 Clear()、Remove()、RemoveAll()、RemoveAt()、RemoveRange()。

和 Find()等方法一样，可以通过 Predicate<T>委托指定要移除的元素需要满足的条件。Remove()等方法的定义如下：

```
public void Clear();
public bool Remove(T item)
public int RemoveAll(Predicate<T> match);
public void RemoveAt(int index);
public void RemoveRange(int index,int count);
```

其中，item 是类型为 T 的要删除的特定元素，注意它只是一个引用。match 是 Predicate<T>类型的委托，用来表示被移除元素需要满足的条件。index 是从 0 开始的要移除元素的索引，count 表示要移除的元素个数。

示例代码 4-12 演示如何使用这些函数从列表 List<T>中移除元素，首先创建一个 List<string>列表对象 strLst，通过 Remove("abc")将"abc"从 strLst 中移除，再通过 RemoveAt(3)移除列表中索引为 3 的元素。通过 RemoveAll(BeginWithA)将 strLst 中移除所有以 A 开始的字符串。通过 RemoveRange(1,3)从 strLst 中移除索引为 1 开始的 3 个元素，最后通过 Clear()移除列表中剩下的所有元素。

示例代码 4-12

```
static void UseRemoves( )
{
    string[] ary = { "begin", "ABC", "abc", "Iaha", "hehe", "Ing", "Big",
    "Aig", "Amer", "end" };                    //准备数据
    List<string> strLst = new List<string>( );
    strLst.AddRange(ary);                      //将数据添加到 List<string>中
    PrintList<string>("原数据", strLst);       //打印数据
    strLst.Remove("abc");                      //移除 abc
    PrintList<string>("第一次", strLst);
    strLst.RemoveAt(3);                        //移除索引为 3 的元素
    PrintList<string>("第二次", strLst);
    strLst.RemoveAll(BeginWithA);              //移除所有以 A 开始的字符串
    PrintList<string>("第三次", strLst);
```

```
    strLst.RemoveRange(1, 3);                //移除从索引为 1 开始的 3 个元素
    PrintList<string>("第四次", strLst);
    strLst.Clear( );                         //移除所有元素
    PrintList<string>("第五次", strLst);
}
```

示例代码 4-12 的输出如下，从中可以看出通过这些方法可以按照开发人员的需要从列表中移除不需要的元素。注意，第 5 次由于所有元素都被移除，所以输出为空白。

```
原数据:begin ABC abc Iaha hehe Ing Big Aig Amer end
第一次:begin ABC Iaha hehe Ing Big Aig Amer end
第二次:begin ABC Iaha Ing Big Aig Amer end
第三次:begin Iaha Ing Big end
第四次:begin end
第五次:
```

注意：RemoveAll()只能移除列表中所有符合条件的元素，而 Clear()是移除列表中任何元素。如果需要移除某个符合条件的元素，可以通过 Find()先找到该元素的索引，然后通过 RemoveAt()方法移除该元素。

4.4.7　使用泛型字典 Dictionary<TKey, TValue>

日常开发中比较常用的数据集合类，除了前面介绍的列表类，还有本节即将介绍的字典类——Dictionary<TKey, TValue>。字典类用来表示一个键值集合，即每个元素包含一个键（Key）和值（Value），其中 Key 是不能相同的，而 Value 则可以相同。如下面的列表中，在同一个学校中所有学生的学号是唯一的，而学生的姓名则可能重复（比如李四），这样学号就是 Key，而姓名就是 Value。

```
学号            姓名
0101            李四
0102            杨武
0203            李四
0203            王二
```

Dictionary<TKey, TValue>就是通过两个泛型参数 TKey 和 TValue 分别表示字典数据集中 Key 和 Value 的数据类型，比如前面的例子可以用 Dictionary<string, string>类型的集合来表示，即学号和姓名都是 string 类型。Dictionary<TKey, TValue>同样提供了大量的成员，便于访问字典中的元素，如表 4-2 所示。

表 4-2　Dictionary<TKey,TValue>主要成员

成员名称	成员说明
Comparer	Public只读属性，获取用于确定字典中的键是否相等的IEqualityComparer<TKey>对象
Count	Public只读属性，获取包含在当前字典中的键/值对的数目，即元素个数
Keys	Public只读属性，获取包含在当前字典中的所有键的集合
Values	Public只读属性，获取包含当前字典中的所有值的集合
Add	Public方法，将指定的键和值添加到当前字典中，键值不能已经存在

续表

成 员 名 称	成 员 说 明
Clear	Public方法，从当前字典中移除所有的键和值
ContainsKey	Public方法，确定当前字典中是否包含指定的键
ContainsValue	Public方法，确定当前字典中是否包含特定值
Remove	Public方法，从当前字典中移除所指定的键的值

从表 4-2 中可以看出，Dictionary<TKey, TValue>提供的成员方法并不多，但是也包含了基本的添加、移除、查询功能。而且它重写了索引器，可以通过中括号“[ ]”和 Key 值来访问该 Key 所对应的值。后面几节将介绍这些成员的具体使用。

4.4.8　添加和访问 Dictionary<TKey, TValue>元素

在使用字典之前，首先要决定字典的两个数据类型：键类型 TKey 和值类型 TValue，这个和具体的应用有关，然后通过 new 运算符创建对应的字典对象。如下面的代码中，intStrDic 是一个键类型为 int，值类型为 string 的字典对象，而 strIntDic 则正好相反，是一个键类型为 stirng，值类型为 int 的字典对象。

```
Dictionary<int, string> intStrDic = new Dictionary<int, string>( );
Dictionary<string, int> strIntDic = new Dictionary<string, int>( );
```

在创建字典对象之后，就可以通过 Add()方法添加一个元素到字典中，字典的每个元素都包含一个键和值，它们是一起添加到字典中且一一对应的。Add()方法的定义如下：

```
public void Add(TKey key, TValue value);
```

其中，第一个参数 key 为键类型 TKey，表示新元素的键，第二个参数 value 为值类型 TValue，表示新元素的值。如下面的代码中，第一句为 intStrDic 字典添加新元素，它的键为 int 类型的 0，值为 string 类型的 Zero，而第二句则为 strIntDic 字典添加一个恰好相反的元素。

```
intStrDic.Add(0, "Zero");
strIntDic.Add("Zero", 0);
```

在添加元素之后，可以通过中扩号“[ ]”和键来访问对应的元素，比如要访问 intStrDic 中键 0 所对应的元素，用如下代码即可。

```
string val = intStrDic[0];
```

字典的索引器（即：[ ]运算符的参数）不是 int 类型，而是 TKey，上一个例子中只是因为 TKey 为 int 而已。比如，要从 strIntDic 中读取 key 为 Zero 元素的值用如下代码：

```
int val = strIntDic["Zero"];
```

由此可见，添加和访问字典中的元素都是非常容易的。值得注意的是，如果添加一个已经存在的键值到字典中会产生异常，所以在添加之前不确定是否存在，就要先通过 ContainsKey()方法判断该键是否已经存在，4.4.9 节将介绍这个方法的使用。

另外，如果需要遍历字典中的所有元素，那么就需要使用 Dictionary<TKey, TValue>的 Keys 和 Values 属性，前者提供当前字典中所有的键集合，后者提供当前字典中的值集

合。它们所返回的类型都是泛型 ICollection<T>的一个实例类型，所以可以通过 foreach 语句遍历它们。所以，通常有两种方法遍历字典中的元素：

- 通过遍历 Keys 属性得到每一个的键 key，然后通过 key 访问它对应的值 value。
- 直接通过 Values 属性得到所有的值 value，但是这样不能得到 value 所对应的键 key。

示例代码 4-13 演示上面提到的两种遍历字典的方法。首先创建一个 Dictionary<int, string>类型的对象 intStrDic，并添加随机产生 5 个元素。然后，通过 foreach 遍历 intStrDic 的 Values 属性获取所有值。最后，通过 foreach 遍历 intStrDic 的 Keys 属性获取所有键，并进一步获取所有值。

示例代码 4-13

```
static void ForEachDic( )
{
    //创建一个新的 Dictionary<int, string>字典对象 intStrDic
    Dictionary<int, string> intStrDic = new Dictionary<int, string>( );
    Random rd = new Random( );       //用来产生随机产生整数
    for (int i = 1; i <= 5; i++)     //为字典对象 intStrDic 添加 5 个元素
    {
        int key = rd.Next(1, 200);
                           //随机产生一个整数，作为元素的 key，其十六进制字符串为值
        intStrDic.Add(key, string.Format("0x{0}", key.ToString("X8")));
    }
    //第一种方法，通过遍历字典的 Values()方法获得字典中所有的元素
    System.Console.WriteLine("第一种:");
    foreach (string val in intStrDic.Values)
    {
        System.Console.Write("{0}  ", val);
    }
    System.Console.WriteLine( );
    //第二种方法，通过便利字典中的 Keys()方法获取元素的键 key，然后通过[] 和 key 获取
      对应的值 value
    System.Console.WriteLine("第二种:");
    foreach (int key in intStrDic.Keys)
    {
        string val = intStrDic[key];
        System.Console.Write("<{0},{1}>  ", key, val);
    }
    System.Console.WriteLine( );
}
```

示例代码 4-13 的输出如下，由于元素是随机产生的，所以每次运行产生的结果会不一样，但是肯定都是 5 个元素，而且第 1 种和第 2 种方法打印出来的值都一样的。

```
第一种:
0x0000003E  0x000000BA  0x0000002F  0x00000018  0x000000B0
第二种:
<62,0x0000003E>   <186,0x000000BA>   <47,0x0000002F>   <24,0x00000018>
<176,0x000000B0>
```

4.4.9　查询和移除 Dictionary<TKey, TValue>元素

在某些应用中，需要判断指定的键或值是否在字典中已经存在，比如在添加一个元素

时就要确保添加的键在字典中不存在。可以通过 Dictionary<TKey, TValue>提供的两个方法 ContainsKey()和 ContainsValue()，来检查指定的键和值是否在字典中已经存在，它们的定义如下：

```
public bool ContainsKey(TKey key);
public bool ContainsValue(TValue value);
```

其中，key 是要查找的键，vlaue 是要查找的值，如果存在则返回 true，否则返回 false。如下面的代码，第一行判断 intStrDic 中为 0 的键是否存在，如果存在 has0 为 true，否则为 false。第二行则判断 intStrDic 中为 Zero 的值是否存在。

```
bool has0 = intStrDic.ContainsKey(0);
bool hasZero = intStrDic.ContainsValue("Zero");
```

通过 ContainsKey()确认元素在字典中存在之后，可以通过 Remove()方法将该元素从字典中移除，当然，也可以通过 Clear()方法将所有元素移除。这两个方法的定义如下：

```
public void Clear();
public bool Remove(TKey key);
```

其中，key 表示要移除的元素的键。如果成功移除指定元素，则 Remove()返回 true，如果元素不存在，则移除失败，返回 false。

示例代码 4-14 演示 Remove()和 Clear()方法的使用。首先，将 intAry 和 strAry 中的元素一对一添加到字典 numDic 中。然后，使用 ContainsKey()和 ContainsValue()判断指定的元素是否存在。最后通过 Remove()和 Clear()移除字典中的元素。

示例代码 4-14

```
static void UseRmvClr( )
{
    int[] intAry = { 1, 2, 3, 4, 5};      //intAry 和 strAry 用来创建字典元素
    string[] strAry = { "one", "two", "three", "four", "five" };
    Dictionary<int, string> numDic = new Dictionary<int, string>( );
    for (int i = 0; i < 5; i++)           //将元素添加到字典 numDic 中
    {
        numDic.Add(intAry[i], strAry[i]);
                               //由于可以保证 key 不重复，所以不用判断是否已经存在
    }
    System.Console.WriteLine("包含键 0? {0}", numDic.ContainsKey(0));
    System.Console.WriteLine("包含键 2? {0}", numDic.ContainsKey(2));
    System.Console.WriteLine("包含值 zero? {0}", numDic.ContainsValue
    ("zero"));
    System.Console.WriteLine("包含值 two? {0}", numDic.ContainsValue
    ("two"));
    numDic.Remove(2);                     //移除键为 2 的元素
    System.Console.WriteLine("包含值 two? {0}", numDic.ContainsValue
    ("two"));
    numDic.Clear( );                      //移除所有元素
    System.Console.WriteLine("剩下元素个数: {0}", numDic.Count);
}
```

示例代码 4-14 的输出如下：

```
包含键 0? False
```

```
包含键 2? True
包含值 zero? False
包含值 two? True
包含值 two? False
剩下元素个数：0
```

4.5　更多高级特性

C# 4.0 还包括不少特性，包括分部类、匿名类型、扩展方法等，并非所有都常用，比如分部类主要用于 Visual Studio 自动生成代码。本节将介绍比较实用的匿名类型和扩展方法。

4.5.1　定义和使用匿名类型

顾名思义，匿名类型是指不需要显式指定类型名称的类，对于临时使用的类型，采用匿名类型可以减少过多的类定义。C#中，匿名类型提供一种方便的方法，可用来将一组只读属性封装到单个对象中，而无须首先显式定义一个类型。

在 C#中，类型名由编译器生成，并且不能在源代码中使用。如示例代码 4-15 所示，var 关键字表示变量 val 是匿名类型，具体类型由编译器自行根据上下文确定。new{}语句用来创建匿名类型，{}里面的内容表示了匿名类型的字段和值。

示例代码 4-15 中，匿名类型包含两个字段 StrVal 和 IntVal，它们的类型，编译器会根据初值自行理解成 string 和 int。后面的代码中，就如同使用普通类一样使用匿名类对象 val 即可，可以通过 val.StrVal、val.IntVal 获取它的字段的值，通过 val.ToString()调用它的函数。

示例代码 4-15

```
static void Main(string[] args)
{
    //定义一个匿名类变量 val，它包含两个字段 StrVal 和 IntVal
    var val = new { StrVal = "a string", IntVal = 10};
    //获取并使用 val 的字段数据
    System.Console.WriteLine("val.StrVal = " + val.StrVal);
    System.Console.WriteLine("val.IntVal = " + val.IntVal.ToString( ));
    //设置匿名类成员的值，错误
    //val.IntVal = 11;
    //使用匿名类的 ToString()方法
    System.Console.WriteLine("val.ToString() = " + val.ToString( ));
}
```

示例代码 4-15 的输出如下：

```
val.StrVal = a string
val.IntVal = 10
val.ToString() = { StrVal = a string, IntVal = 10 }
```

C#中，匿名类型通常表示临时使用的只读数据，所以必须在创建的时候初始化各字段

的数据，并且只能读取这些字段的值，不能设置这些字段的值。如示例代码 4-16 中“val.IntVal=10;”试图修改 IntVal 字段的值，就会产生错误。

提示：合理使用匿名类型可以很大程度上减少不必要的类定义，增加代码的可读性。另外，匿名类型常用在 LINQ 中保存查询结果，将在第 15 章详细介绍。

4.5.2　添加扩展方法扩展现有类

扩展方法是 C# 4.0 的又一个特性，它允许开发人员在不创建派生类型和不修改原始类型的基础上，直接向现有类型“添加”方法。扩展方法是一种特殊的静态方法，但可以像扩展类型上的实例方法一样进行调用。

包含扩展方法的类必须定义为静态（static）类，扩展方法也必须定义为静态（static）方法，它的第一个参数指定该方法被扩展到哪一个类型中，同时第一个参数要用 this 关键字修饰，扩展方法可以作为普通方法一样的方式被调用。

示例代码 4-16 演示扩展方法的使用。包含扩展方法的类 ExtendMethods 被定义为 static 类，扩展方法：IsRight()、PrintHint()都被定义为 static 方法，而且第一个参数都用 this 关键字修饰，表示该方法扩展到某个类型。其中，IsRight()方法被扩展到 string 和 int 类型，而 PrintHint()方法则被扩展到 object 类型，这样，PrintHint()就可以被所有类型访问，因为所有类型都是 object 类型的子类。

示例代码 4-16

```
public static class ExtendMethods
{
    //定义 string 类型的扩展方法 IsRight()
    public static bool IsRight(this string str)
    {
        switch (str.ToUpper( ).Trim( ))
        {
            case "RIGHT":
                return true;
            case "YES":
                return true;
            case "OK":
                return true;
            default:
                return false;
        }
    }

    //定义一个 int 类型的扩展方法 IsRight()
    public static bool IsRight(this int val)
    {
        if (val > 10)
        { return true; }
        else
        { return false; }
    }
```

```
    //定义一个对于所有类型的扩展方法，打印类信息同时添加一个字符串作为提示信息
    public static void PrintHint(this object obj, string hint)
    {
        System.Console.WriteLine(obj.ToString( ) + "--" + hint);
    }
}

namespace UseExtMethod
{
    //必须显式声明使用 ExtendMethods 命名空间
    using ExtendMethods;
    class Program
    {
        static void Main(string[] args)
        {
            //使用 string 类的扩展方法 IsRight()
            string str1 = "right", str2 = "err";
            System.Console.WriteLine("str1.IsRight() = " + str1.IsRight( ).
            ToString());
            System.Console.WriteLine("str2.IsRight() = " + str2.IsRight( ).
            ToString( ));

            //使用 int 类型的扩展方法 IsRight()
            int i1 = 11, i2 = 5;
            System.Console.WriteLine("i1.IsRight() = " + i1.IsRight( ).
            ToString( ));
            System.Console.WriteLine("i2.IsRight() = " + i2.IsRight( ).
            ToString( ));
            //使用 Object 类型的扩展方法 PrintHint()
            string str3 = "Jone";
            str3.PrintHint("你好！");
        }
    }
}
```

生成并运行示例代码 4-16，得到程序输出如下：

```
str1.IsRight() = True
str2.IsRight() = False
i1.IsRight() = True
i2.IsRight() = False
Jone--你好！
```

在使用扩展方法前需要显式引用扩展方法所在的命名空间，如示例代码 4-16 中的代码“using ExtendMethods;”。另外，在扩展方法被调用时和普通的成员方法调用一样，扩展方法的第一个参数也需要在调用时指出。

4.6　小　　结

C#作为一门面向对象高级编程语言，不仅支持类和接口，还支持通过委托实现函数的动态配置，并在此基础上搭建了事件机制，使得 Windows 窗体开发更加灵活方便。另外，

C#还支持泛型开发，并提供自带的集合类，通过这些类使得数据在内存中的表现形式更加丰富并且易于操作。

本章通过实例，全方位介绍了 C#中这些高级特性，通过本章的学习，读者应该掌握以下知识点：

- ❑ 什么是委托？委托应该如何定义？
- ❑ 委托该如何调用？委托链上的函数如何执行？
- ❑ 如何定义事件和引发事件？
- ❑ 如何订阅和处理事件？
- ❑ 如何定义和使用泛型？
- ❑ .NET 类库提供了那些集合类？
- ❑ 如何使用 List<T>列表集合类？
- ❑ 如何使用 Dictionary<TKey, TValue>字典集合类？
- ❑ 匿名类型如何使用？
- ❑ 扩展方法如何使用？

第 2 篇　开发应用程序

第 5 章　Windows 窗体程序

窗体应用程序是.NET 下开发 Windows 窗体用户界面最基本和常用的技术，窗体可以为用户提供非常简单而友好的操作界面，如文本框输入数据、列表中选择数据、报表、网格显示数据等。通过这些窗体控件的有机结合，以及它们的各种事件处理函数和后台代码结合，可以开发出复杂的应用逻辑。本章将介绍.NET 下通过 WinForm 技术开发 Windows 窗体程序的常见知识。

5.1　第一个窗体应用程序

在 Visual Studio 2010 中，通过 Windows 窗体程序模板构建具有窗体（Form）的用户界面程序的基本框架，包括一个主界面、启动代码等。通过窗体设计器可以可见即可得的方式设计窗体，使得开发更加轻松快捷。本节将简单介绍窗体应用程序的产生和运行。

5.1.1　创建和运行窗体程序

在 C#中，开发窗体应用程序是一个将窗体、控件、应用逻辑三者合理集成的过程。首先需要选择窗体和控件，对控件进行布局，然后根据应用逻辑添加窗体和控件的事件响应函数，并添加适当的应用逻辑代码。

本节将创建一个简单的 Windows 窗体应用程序——InputLog，它可以临时记录用户的输入，并显示到一个列表中。通过 Visual Studio 2010 创建 Windows 窗体应用程序，通常需要以下 4 个步骤：

（1）运行 Visual Studio 2010 开发环境。

（2）通过菜单“文件”|“新建”|“项目”，打开“新建项目”窗体，如图 5-1 所示。在项目类型中选择 Visual C#下的 Windows 选项，在“模板”列表中选择“Windows 窗体应用程序”选项。

（3）输入应用程序的名称等属性，在“名称”文本框中输入应用程序名称，在该实例中输入 InputLog，在“位置”文本框中输入应用程序源代码要保存的路径，如果输入的路径不存在，会自动创建该路径。

（4）单击“确定”按钮完成 Windows 应用程序 InputLog 的创建。

Visual Studio 会根据选择的模板——Windows 窗体应用程序生成窗体应用程序的基本框架，包括一个主窗体，启动程序代码等，在 5.1.2 节中将详细介绍。此时，应用程序 InputLog 已经可以运行，并且具有一个窗体，但它还只是一个空白的窗体，还没有任何实际意义。通过菜单“调试”|“开始执行”运行程序，可以得到一个空的标题为 Form1 的窗体，可以

最大化、最小化和关闭窗体。



图 5-1　创建 InputLog 界面

5.1.2　分析窗体应用程序的结构

Visual Studio 根据应用程序模板自动生成的窗体应用程序 InputLog 的基本框架，从“解决方案资源管理器”中可以看到，它主要包括以下 3 部分。

- 主窗体 Form1：自动创建的默认的主窗体，继承自 Form 类，窗体内没有添加任何控件。
- 启动程序 Program：带有 Main()入口函数的类，应用程序就从 Program.Main 启动。
- 程序信息 AssemblyInfo：程序的版本、版权等信息。

示例代码 5-1 给出窗体 Form1 和启动类 Program 的主要代码，可以看出，Form1 类只是简单地从 Form 类继承，然后重写默认构造函数，并通过 InitializeComponent()函数布局窗体控件。Program 类是一个静态类，它提供一个应用程序入口函数 Main()，在 Main()函数中通过“Application.Run(new Form1())”代码，创建一个 Form1 窗体，并作为主窗体运行应用程序。最后一个部分是 AssemblyInfo.cs 的代码，开发人员根据需要在对应的位置填入公司名称、版本号、产品名称、版权等信息，生成之后的 EXE 就拥有这些信息。

示例代码 5-1

```
public partial class Form1 : Form
{
    public Form1( )
```

```
    {
        InitializeComponent( );
    }
}
static class Program
{
    //应用程序的主入口点
    [STAThread]
    static void Main( )
    {
        Application.EnableVisualStyles( );
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1( ));
    }
}
//有关程序集的常规信息通过下列属性集
//控制、更改这些属性值可修改
//与程序集关联的信息
[assembly: AssemblyTitle("InputLog")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("WwW.YlmF.CoM")]
[assembly: AssemblyProduct("InputLog")]
[assembly: AssemblyCopyright("Copyright © WwW.YlmF.CoM 2008")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//将 ComVisible 设置为 false 使此程序集中的类型
//对 COM 组件不可见。如果需要从 COM 访问此程序集中的类型
//则将该类型上的 ComVisible 属性设置为 true
[assembly: ComVisible(false)]

//如果此项目向 COM 公开，则下列 GUID 用于类型库的 ID
[assembly: Guid("c9bb9e4f-36ca-4c8f-b6da-63b66f0965f4")]

//程序集的版本信息由下面四个值组成:
//主版本
//次版本
//内部版本号
//修订号
//可以指定所有这些值，也可以使用“内部版本号”和“修订号”的默认值
//方法是按如下所示使用“*”:
//[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
```

5.1.3　用窗体设计器编辑控件和窗体

在 5.1.2 节创建的默认窗体应用程序，还只是一个空白窗体，甚至连窗体的标题都还是默认值 Form1，这显然不能满足需要。接下来就需要通过 Visual Studio 2010 提供的窗体设计器为窗体添加及编辑控件，设置控件和窗体的属性及事件等。

窗体控件是.NET 下与用户进行交互的最基本的可视化单元，它们可以添加到窗体或其他容器类控件上，通过不同控件的组合和布局、不同外观属性和事件处理逻辑，可以开发出各种复杂的应用程序界面。.NET 类库提供了多种类型的控件，包括用于显示的文本框、

按钮、下拉列表框、单选按钮、网页控件等。另外，通过 ToolStrip 和 MenuStrip 控件，还可以创建包含文本和图像、显示子菜单及承载其他控件（如文本框和组合框）的工具栏和菜单。

要实现特定的用户界面，需要添加控件到窗体上。在 Visual Studio 2010 中，为窗体添加控件相当简单，只需要通过以下 4 个步骤即可：

（1）通过在“解决方案资源管理器”中双击要编辑的窗体，在窗体设计器中打开窗体。本例中双击“Form1.cs”，显示出窗体 Form1 可见即所得的编辑界面。

（2）通过菜单“视图”|“工具箱”打开工具箱视图，这里列出了当前选中窗体上可以使用的所有控件，这些控件按照分组列出。

（3）从“工具箱”选择需要添加的控件，通过鼠标拖放到 Form1 界面上，就可从设计器上看到控件的具体效果。

（4）选中控件，通过“属性管理器”窗体，修改控件的属性，包括前景色、字体、背景色、大小、位置等多种外观信息，并且修改后可以马上更新到设计器中。

在本例中，从工具箱中拖放 1 个 Button 和 2 个 TextBox 控件（都在公共控件栏中）到 Form1 界面，分别命名为 btnSubmit、tbInput 和 tbLog，并放在适当的位置，如图 5-2 所示，具体的属性设置如表 5-1 所示。

表 5-1　InputLog控件和属性列表

控　件	属　性　名	值	说　明
Form1	Text	输入记录实例	修改标题
	FormBorderStyle	FixedSingle	窗体 Form1 为尺寸不可变
	MaximumuBox	false	Form1 的最大化按钮不可用
btnSubmit	Text	提交	修改按钮文本
tbLog	BackColor	Black	黑色背景
	ForeColor	White	黑色前景色
	Text	这里是记录	设置默认的输入信息
	ReadOnly	true	该文本框为自读
tbInput	Text	这里输入值	文本框 tbResult 的默认文本
	ReadOnly	false	文本框可以输入数据



图 5-2　InputLog 设计图

从图 5-2 中可以看出，窗体上文本框的颜色正如设置的一样为黑底白字。属性管理器是窗体程序在进行界面设计的主要工具之一。除此之外，Visual Studio 的“布局”工具栏也提供很多有用的设计工具，比如控件左对齐、右对齐、相同大小、Tab 键顺序设置等。由于篇幅限制，这里不做详细介绍，读者可以创建一些窗体，多做试验。

5.1.4　添加窗体后台逻辑代码

在 5.1.3 节只是将控件添加到界面上，从本质上说还是一个空壳，并没有实际的逻辑代码，接下来就要为应用程序 InputLog 添加逻辑代码。通常一个窗体的逻辑代码可以分为以下几部分。

- 控件事件处理函数：这些函数本质上是窗体的成员，但通常是为了响应用户的界面操作，比如单击按钮、在文本中输入文字等。本节将介绍如何添加控件事件处理函数。
- 窗体事件处理函数：这些函数本质上也是窗体的成员，但通常是对窗体的生命周期进行监视，并给出相应的处理，在 5.2 节将详细介绍有关知识。
- 窗体类成员：这些函数是窗体类的普通成员，它们是开发人员根据实际应用需要为窗体类添加的成员，可以是字段、属性和方法，也可以是事件和委托等。这些与为普通类添加成员一样，不再赘述。

在 Visual Studio 2010 中，通过窗体设计器和属性管理器界面可以方便地为控件添加事件响应函数，主要步骤如下：

（1）通过在“解决方案资源管理器”中双击要编辑的窗体，在窗体设计器中打开窗体。本例中双击 Form1.cs，显示出窗体 Form1 可见即所得的编辑界面。

（2）选择要添加事件的控件或窗体，在右键快捷菜单中选择“属性”命令，进入属性管理器。选择事件视图，在视图中双击要添加响应函数的事件即可。本例中，选中按钮 btnSubmit，并为它添加 Click 事件处理函数 btnSubmit_Click()，方法名是 Visual Studio 自动生成，可以手动更改。

示例代码 5-2 给出了实例 InputLog 中窗体 Form1 的主要代码，其中 partial 关键字表示 Form1 是分部类，这里只是给出了它的部分代码，这是因为 Form1 的另外一部分代码通过 Visual Studio 自动生成和维护，通常不需要修改。btnSubmit_Click()方法是按钮 btnSubmit 的 Click 事件的响应函数，它从文本输入框 tbInput 取得用户输入的文本 input，然后将 input 通过 AddInputToLog()方法将它显示到只读文本框 tbLog 中，最后清空输入框 tbInput，等待用户新的输入。图 5-3 是该 InputLog 的运行效果。

示例代码 5-2

```
public partial class Form1 : Form
{
    public Form1( )                                   //默认构造函数
    {
        InitializeComponent( );
    }
    //将新的输入添加到 log 文本框 tbLog 中
    private void AddInputToLog(string input)
    {
```

```
        this.tbLog.AppendText("\r\n"+input);      //追加 input 到 Log 文本框
        this.tbLog.ScrollToCaret();               //滚到 Log 文本框到最后一行
    }
    //“提交”按钮 Click 事件处理函数，通过 AddInputToLog()方法将用户输入的问题添加
    到 Log
    private void btnSubmit_Click(object sender, EventArgs e)
    {
        string input = this.tbInput.Text;         //获取当前输入的文本
        this.AddInputToLog(input);                //将输入的文本添加到日志
        this.tbInput.Clear();                     //清空文本输入框
    }
}
```

注意：Form1()的构造函数必须调用 InitializeComponent()方法，该方法用来绘制窗体界面，由 Visual Studio 生成的 Form1 的另外一部分代码中。Visual Studio 自动生成的事件响应函数名称通常为 ControlName_EventName，可以手动修改，但一般不需要修改。



图 5-3　InputLog 运行效果

5.2　深入学习 Windows 窗体

窗体应用程序中最基本的一个元素是窗体——Form，它是装载和显示其他控件的容器，也是可视化用户界面的基础。.NET 类库中 Form 类提供了丰富的属性来设置窗体的外观，提供丰富的事件使得开发人员可以控制窗体的整个生命周期。本节将深入介绍窗体的生命周期和主要属性。

5.2.1　了解 Windows 窗体生命周期

在 WinForm 应用程序中，任何窗体都是从.NET 类库的 System.Windows.Forms.Form 类派生而来，Form 封装了所有窗体都具备的属性、方法和事件。任何一个窗体（Form）都要经历一个创建→显示→使用→关闭→销毁的过程，这一系列过程被称为窗体的生命

周期。

在窗体生命周期的各个阶段，Form 类都会引发对应的事件，通过处理这些事件，开发人员可以很好地控制窗体的每一个过程。如图 5-4 所示，在窗体生命周期的各阶段都有不同的事件和适当的工作。

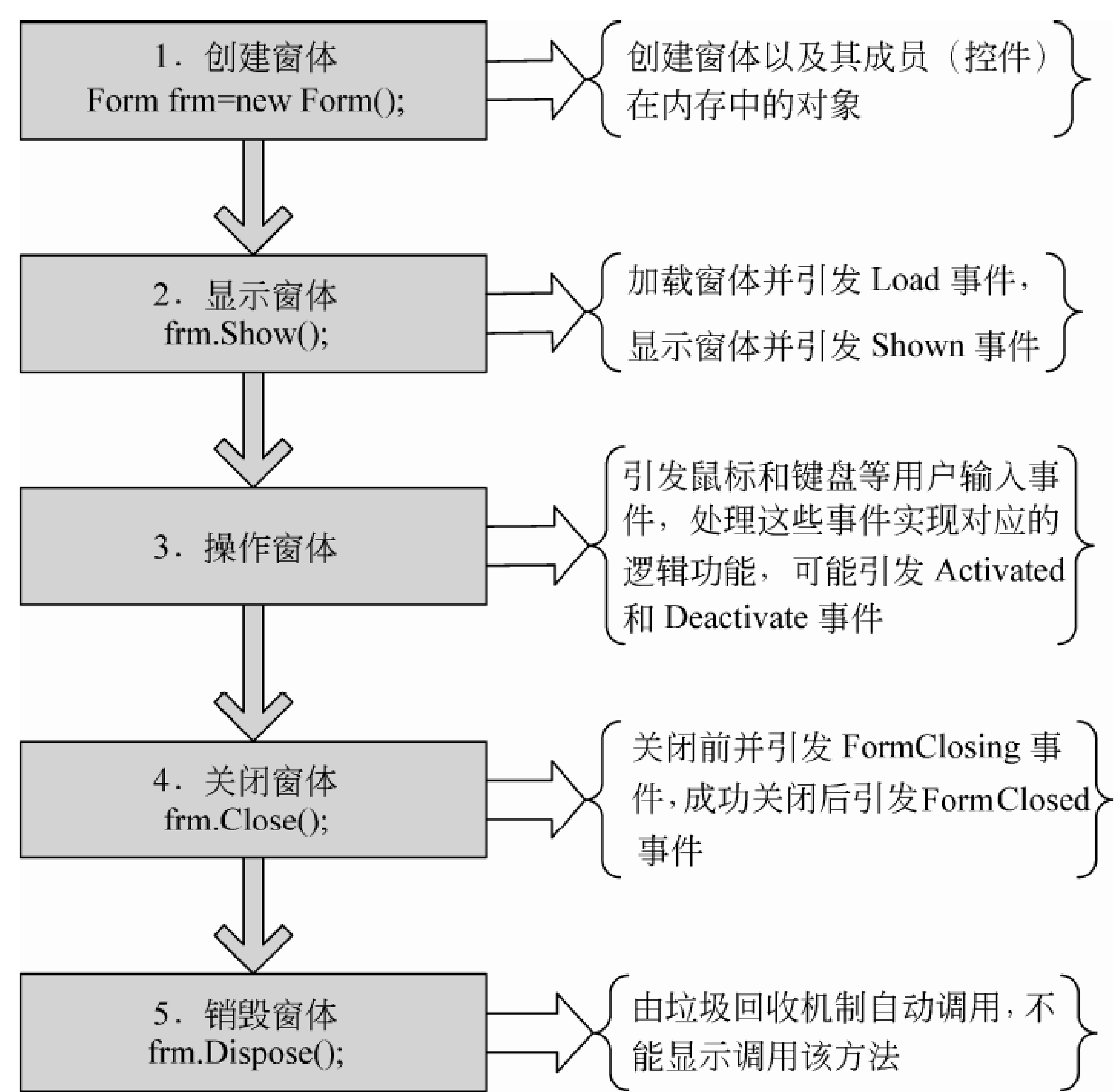


图 5-4　窗体生命周期各阶段

1．创建窗体（new）

窗体作为一个普通的类，需要创建具体实例才能进行相关操作，所以首先就要创建新的窗体实例，通过 new 关键字调用构造函数为窗体创建新实例；在构造函数中常常做一些数据初始化操作，同时创建该窗体下的所有控件实例。

2．显示窗体（Show/ShowDialog）

在窗体实例创建之后，需要通过调用窗体的 Show()或 ShowDialog()成员函数显示它。在窗体第一次显示时，会加载窗体及其所有控件，并产生 Load 事件。通常，可以通过处理 Load 事件来完成整体的界面和数据初始化，比如默认标题设置，数据库连接配置等。窗体成功显示之后，还会引发 Shown 事件。

3．使用窗体

在窗体成功显示之后，窗体就需要和用户发生交互，窗体和用户之间的交互主要是通过控件事件和窗体事件的响应函数来实现。主要包括鼠标事件、键盘事件、界面外观变化事件、后台数据变化事件几大类。当然需要增加哪些方法、处理哪些事件，以及实现功能和处理逻辑，都是和具体的窗体功能有关的。在操作窗体时，当窗体失去焦点（变为后台

窗体）时会引发 Deactivate 事件，当窗体重新获得焦点（变为前台窗体）时会引发 Activated 事件。

4. 关闭窗体（Close）

在具体操作完成后，可以通过窗体的 Close()成员函数关闭窗体。在窗体关闭前会引发 FormClosing 事件，在该事件的处理函数中可以完成一些关闭前的操作，比如窗体关闭确认、文件关闭、资源释放等。在窗体成功关闭之后，会引发 FormClosed 事件，可以在这里做最后的数据处理，做最后的资源释放，或者一些特定的需求，比如释放窗体占用的打印机；根据需要显示新的窗体等。

5. 销毁窗体（Dispose）

在窗体成功关闭之后，其实窗体对象所在内存并没有释放，该对象仍然存在与内存中，真正对象销毁是在.NET 进行垃圾回收时才销毁，在窗体对象销毁的时候就会调用窗体的 Dispose()成员函数。这也是对窗体进行操作的最后机会，实际开发中很少使用该函数来进行资源释放等操作，因为该函数的调用时间是不可预测的。

在窗体的整个生命周期中，除了前面提到的几个关键事件之外，还包括在与用户交互时产生的鼠标和键盘等事件，在某些窗体属性发生更改时也会产生事件。由于篇幅关系，这里不一一介绍这些事件，表 5-2 列出了 Form 类的主要事件，读者可以查阅 MSDN 或者微软官方网站获取更多信息。

表 5-2　窗体（Form类）的主要事件

事 件 名 称	事 件 说 明
Layout	当窗体要对其他控件重新布局时引发的事件
MidChildActive	当窗体作为多文档程序父窗体时，它所包含的子窗体被激活时引发的事件
Move	当窗体被移动时引发的事件
Resize	当窗体尺寸发生变化时引发的事件
SizeChanged	窗体的尺寸变化时引发的事件，Size 属性变化时引发
LocationChanged	窗体显示的坐标发生变化时引发的事件，Location 属性变化时引发
Paint	当窗体用户区域重新绘制时引发的事件
FontChanged	窗体的字体发生变化时引发的事件，Font 属性改变时引发
ForeColorChanged	窗体的前景色变化时引发的事件，ForeColor 属性改变时引发
BackColorChanged	窗体的背景色变化时引发的事件，BackColor 属性改变时引发
TextChanged	窗体标题变化时引发的事件，Text 属性改变时引发
CursorChanged	窗体的光标发生变化时引发的事件，Cursor 属性改变时引发
Load	窗体第一次加载是引发的事件
Shown	窗体每次显示时都会引发的事件，进行一些绘制的特殊处理
FormClosing	窗体关闭前引发的事件，通过该事件可以取消关闭操作
FormClosed	窗体关闭后引发的事件，释放窗体资源最后的机会
Activated	窗体被激活（即作为活动窗体）时引发的事件
Deactivated	窗体失去焦点（即不再是活动窗体）时引发的事件
VisibleChanged	窗体的可见性变化时引发的事件，Visible 属性改变时引发
EnableChanged	窗体的有效性变化时引发的事件，Enable 属性改变时引发

5.2.2　学习 Windows 窗体主要属性

一个窗体通常具有很多重要属性，用来定义和描述窗体的外观和行为，包括窗体的标题、字体、颜色、窗体状态（最大化、最小化等）、位置、宽度、高度、图标、是否置顶等。在.NET 类库中，Form 类提供了丰富的属性来定义和约束窗体的外观，Form 类的主要属性如表 5-3 所示。

从表 5-3 中可以看出，Form 类提供的属性中包括丰富的外观和布局属性，比如 Font、BackColor、Text、Size、Location、Padding 等。也包括了大量和窗体行为相关的属性，比如 TopMost、Cursor、ControlBox、MaximumBox、Enable 等。可见，Form 类根据需要进行属性设置，可以创建出各种各样非常复杂而美观的窗体，以满足开发中的各种需求。

表 5-3　窗体（Form类）的主要属性

属 性 名　称	属 性 说 明
AutoScroll	当窗体大小不够时，是否自动显示滚动条
AutoSize	当窗体包含内容变化时，是否自动调整自身的大小以适应变化
AutoSizeMode	自动调整的模式，当 AutoSize 为 true 时有效
Location	窗体左上角相对于其容器所在的位置，包括（X，Y）坐标
StartPosition	窗体第一次显示时的位置，可以是屏幕中心、父窗体中心、默认位置等，常用父窗体中心和屏幕中心两种
MinimumSize	通过鼠标拖放可以达到的窗体的最小尺寸，包括宽和高，像素为单位
MaximumSize	通过鼠标拖放可以达到的窗体的最大尺寸，包括宽和高，像素为单位
Size	窗体的默认尺寸，包括宽和高，像素为单位
Padding	窗体显示区域和边框之间的间隔，包括上下左右 4 个方向，像素为单位
WindowState	窗体第一次显示时的状态，包括最大化、最小化、正常 3 个选择
Text	窗体的标题
Font	窗体中控件上显示的默认字体
BackColor	窗体的背景颜色
BackgroundImage	窗体背景图片
BackgroundImageLayout	窗体背景图片的布局方式，可选平铺、拉伸、居中等
Cursor	窗体上使用的鼠标光标，可选任何系统提供的鼠标光标
FormBorderStyle	窗体边框样式，包括可变大小、3D、对话框等
UseWaitCursor	窗体鼠标光标是否只使用等待光标
Icon	获取或设置窗体的图标，即窗体标题栏最左边的图标
ControlBox	是否需要窗体标题栏的系统按钮，最大化、最小化、关闭等
MaximumBox	窗体的最大化按钮是否可用，ControlBox 为 true 时有效
MinimumBox	窗体的最小化按钮是否可用，ControlBox 为 true 时有效
ShowIcon	是否显示窗体标题栏最左边的图标，ControlBox 为 true 时有效
ShowInTaskbar	窗体显示时标题是否显示在 Windows 任务栏
TopMost	获取或设置当前窗体是否为最前端显示窗体
Enable	窗体是否可用，如果不可用则鼠标键盘等事件都无法响应
ContextMenuStrip	窗体的右键弹出菜单，该菜单是从已有菜单选择的

通常有两种方法修改一个窗体的属性，一种是在设计时通过“属性管理器”设置窗体

的属性，这样在运行的时候窗体的大部分属性都保持不变。另一种是通过 Form 类的实例在代码中根据实际需要动态修改窗体的属性，这样使得窗体变得更加灵活，在 5.2.3 节中将介绍这种方法。

5.2.3　设置 Windows 窗体的主要属性

通常情况下，只需要简单地在窗体中设置属性即可，但是在某些特殊情况下，需要通过代码来灵活地修改窗体属性，这样可以使用户界面更加美观和友好。在 C#中，只需要通过设置表 5-3 中给出的属性就可以轻松实现这样的功能，关键在于找到合适的时机设置这些属性。

如示例代码 5-3 所示，btnSetProp_Click()是窗体 FrmMain 上的一个按钮 btnSetProp 的 Click 事件处理函数，该应用程序的功能是，在单击按钮 btnSetProp 之后重新设置窗体的标题（Text）、背景色（BackColor）、边框（FormBorderStyle）、宽度（Width）、高度（Height）。并且取消窗体的最大化或最小化状态，返回到正常状态，设置窗体一直置顶。

示例代码 5-3

```
public partial class FrmMain : Form
{
    public FrmMain( )
    {
        InitializeComponent( );
    }
    private void btnSetProp Click(object sender, EventArgs e)
    {
        this.Text = "测试对话框";                    //设置标题
        this.FormBorderStyle = FormBorderStyle.FixedDialog;
                                                    //设置边框为对话框样式
        this.BackColor = Color.Gray;                //灰色背景
        this.WindowState = FormWindowState.Normal;  //窗体正常状态，取消最大化
                                                      和最小化
        this.MinimizeBox = false;                   //不需要最小化按钮
        this.Height = 200;                          //高度设置为 200px
        this.Width = 400;                           //宽度设置为 400px
        this.TopMost = true;                        //始终置顶
   }
}
```

如图 5-5 是设置属性之前窗体 FrmMain 的截图，而图 5-6 是设置属性（调用 btnSetProp_Click()）之后窗体 FrmMain 的截图，可见完全是希望的结果。

5.2.4　显示和关闭 Windows 窗体

在.NET 类库中，Form 类不仅提供了属性和事件，还提供了丰富的方法，但是实际开发中常用的方法如表 5-4 所示。其中 Show()方法是以异步的方式显示窗体，新窗体和当前窗体独立，不会影响当前窗体的用户交互。而 ShowDialog()方法是以同步的方式显示窗体（对话框的方式），该函数会等到新的窗体被关闭之后才会返回，所以会阻塞当前窗体与用

户的交互。



图 5-5　设置属性前的 FrmMain

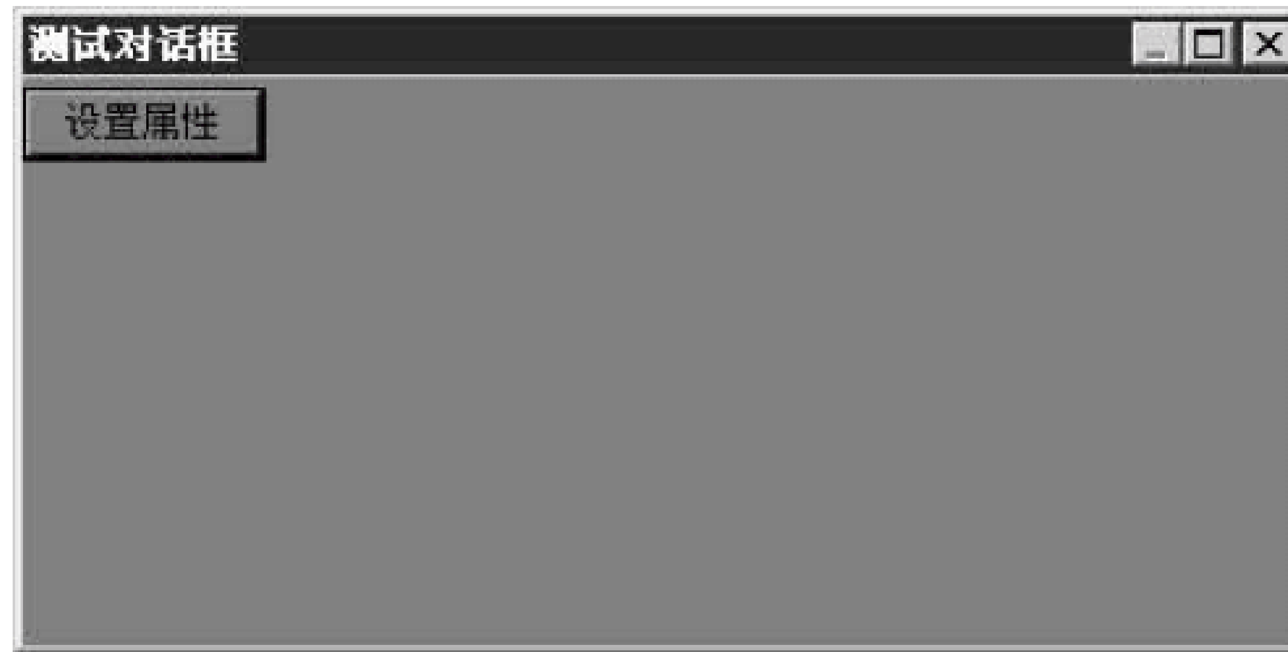


图 5-6　设置属性后的 FrmMain

表 5-4　窗体（Form类）的主要方法

方 法 名 称	方 法 说 明
Show	以普通方式显示窗体，不阻塞主窗体的用户交互
ShowDialog	以对话框的方式显示窗体，阻塞主窗体的用户交互
Close	关闭窗体
Activate	激活窗体

示例代码 5-4 演示如何创建并显示对话框，其中，btnCreate_Click()是按钮 btnCreate 的 Click 事件处理函数。首先判断当前窗体对象_CurrFrm 是否存在，如果存在则通过 Active()方法激活，否则通过 new 运算符新建一个窗体对象。BtnClose_Click()是按钮 btnClose 的 Click 事件处理函数，如果当前窗体存在，通过 Close()方法关闭该窗体。

示例代码 5-4

```
//表示当前创建的窗体对象
private FrmMain  CurrFrm = null;
//创建窗体按钮 Click 事件处理函数
private void btnCreate_Click(object sender, EventArgs e)
{
    if(this._CurrFrm == null)            //如果当前窗体不存在，则新建
    {
        this. CurrFrm = new FrmMain( );
    }
    this._CurrFrm.Show( );               //显示窗体
}
//关闭窗体按钮 Click 事件处理函数
private void btnClose Click(object sender, EventArgs e)
{
    if (this. CurrFrm != null)           //如果窗体存在，则关闭
    {
        this._CurrFrm.Close( );          //关闭窗体
        this. CurrFrm = null;
    }
}
```

5.3　使用常用 Windows 控件

.NET 类库除了窗体外，还提供了很多比较常用的控件，比如前面用到的按钮和文本框，还包括下拉框、列表框、分组容器、Tab 页、网格等。本节将介绍几个比较常用的控件。

5.3.1　Windows 控件共有特性

在.NET 类库中，System.Windows.Forms 命名空间提供各种常用的控件类，使用这些控件类，可以创建丰富的用户界面。一些控件用于在应用程序内进行数据输入，如 TextBox 和 ComboBox。一些控件显示应用程序数据，如 Label 和 ListView。还有些用于在应用程序中调用命令的控件，如 Button。WebBrowser 控件可以在 Windows 窗体应用程序中显示和操作 HTML（Hypertext Markup Language，超文本标识语言）页面。此外，MaskedTextBox 控件是一个高级数据输入控件，允许定义可自动接受或拒绝用户输入的掩码。也可以通过从 UserControl 类派生而创建自己的控件。

所有这些控件，都是从 System.Windows.Forms.Control 类派生而来。Control 类提供了向用户显示信息和从用户获取信息的可视化元素的最基本功能。它处理用户通过键盘和鼠标等输入设备所进行的输入，还处理消息路由和安全。但是 Control 类并不实现真正的界面绘制，它只是定义控件的边界（其位置和大小），提供用于绘制的窗口句柄。总地来说，Control 类通过大量的成员来提供和实现下列任务，以便在 Windows 窗体应用程序中提供可视显示。

- 公开窗口句柄。
- 管理消息路由。
- 提供鼠标和键盘事件，以及许多其他用户界面事件。
- 提供高级布局功能。
- 包含特定于可视显示的许多属性，如 ForeColor、BackColor、Height 和 Width。
- 为 Windows 窗体控件充当 Microsoft ActiveX 控件提供必需的安全和线程支持。
- 视化界面的外观和行为进行设置和控制。

表 5-5 列出了 Control 类提供的常用属性，表 5-6 列出了 Control 类提供的常用方法，表 5-7 列出了 Control 类的常用事件，通过这些使用成员可以开发出非常复杂、友好的用户界面。

表 5-5　Control类主要属性

属 性 名 称	属 性 说 明
Name	获取或设置控件的名称，唯一标志某个控件
Text	获取或设置与此控件关联的文本，不同控件，表示不同的意义；如 Button 的显示文本，Form 窗体的标题，TextBox 的编辑文本等
Tag	获取或设置包含有关控件的数据对象；一个 Object 类，常用来保存当前控件所表示的数据对象
BackColor	获取或设置控件的背景色
ForeColor	获取或设置控件的前景色，也就是字体颜色、线条颜色等

续表

属 性 名 称	属 性 说 明
Font	获取或设置控件显示文字的字体
RightToLeft	获取或设置一个值，该值指示是否将控件的元素对齐，以支持使用从右向左的字体的区域设置
BackgroundImage	获取或设置在控件中显示的背景图像
BackgroundImageLayout	获取或设置 ImageLayout 枚举中定义的背景图像布局
Top	获取或设置控件上边缘与其容器的工作区上边缘之间的距离（以像素为单位）
Bottom	获取控件下边缘与其容器的工作区上边缘之间的距离（以像素为单位）
Left	获取或设置控件左边缘与其容器的工作区左边缘之间的距离（以像素为单位）
Right	获取控件右边缘与其容器的工作区左边缘之间的距离（以像素为单位）
Height	获取或设置控件的高度
Width	获取或设置控件的宽度
MaximumSize	获取或设置大小，该大小是 GetPreferredSize 可以指定的上限
MinimumSize	获取或设置大小，该大小是 GetPreferredSize 可以指定的下限
Size	获取或设置控件的高度和宽度
Location	获取或设置该控件的左上角相对于其容器的左上角的坐标
Bounds	获取或设置控件（包括其非工作区元素）相对于其父控件的大小和位置（以像素为单位）
ClientRectangle	获取表示控件工作区的矩形
ClientSize	获取或设置控件工作区的高度和宽度
DisplayRectangle	获取表示控件显示区域的矩形
AllowDrop	获取或设置一个值，该值指示控件是否可以接受用户拖放到它上面的数据
Anchor	获取或设置控件绑定到的容器边缘，并确定控件如何随其父级一起调整大小
Dock	获取或设置哪些控件边框停靠到其父控件并确定控件如何随其父级一起调整大小
Created	获取一个值，该值指示控件是否已经创建
Capture	获取或设置一个值，该值指示控件是否已捕获鼠标
CanSelect	获取一个值，该值指示是否可以选中控件
CanFocus	获取一个值，该值指示控件是否可以接收焦点
Focused	获取一个值，该值指示控件是否有输入焦点
ContainsFocus	获取一个值，该值指示控件或它的一个子控件当前是否有输入焦点
Handle	获取控件绑定到的窗口句柄
Parent	获取或设置控件的父容器
HasChildren	获取一个值，该值指示控件是否包含一个或多个子控件
Controls	获取包含在控件内的控件集合
Cursor	获取或设置当鼠标指针位于控件上时显示的光标
UseWaitCursor	获取或设置一个值，该值指示是否将等待光标用于当前控件及所有子控件
ContextMenu	获取或设置与控件关联的快捷菜单（右键菜单）
ContextMenuStrip	获取或设置与控件关联的 MenuStrip 菜单控件（右键菜单）
Enabled	获取或设置一个值，该值指示控件是否可以对用户交互作出响应
Visible	获取或设置一个值，该值指示是否显示该控件

表 5-6　Control类的常用方法

方 法 名 称	方 法 说 明
BeginInvoke	在创建控件的基础句柄所在线程上，异步执行委托
Invoke	在拥有此控件的基础窗口句柄的线程上执行委托，通常用于异步函数的调用
EndInvoke	检索由传递的 IAsyncResult 表示的异步操作返回值
Show	对用户显示控件
Hide	对用户隐藏控件
Update	使控件重绘其工作区内的无效区域
Refresh	强制控件使其工作区无效并立即重绘自己和任何子控件
Select	激活控件
DoDragDrop	开始拖放操作
Focus	为控件设置输入焦点
FindForm	检索控件所在的窗体
Scale	缩放控件和任何子控件
PerformLayout	强制控件将布局逻辑应用于子控件
Dispose	释放由 Control 使用的所有资源，窗体对象在回收时会被调用
GetPreferredSize	检索可以容纳控件的矩形区域大小
SetBounds	设置控件的边界
PointToClient	将指定屏幕点的位置计算成工作区坐标
PointToScreen	将指定工作区点的位置计算成屏幕坐标
RectangleToClient	计算指定屏幕矩形的大小和位置（以工作区坐标表示）
RectangleToScreen	计算指定工作区矩形的大小和位置（以屏幕坐标表示）

表 5-7　Control类常用事件

事 件 名 称	事 件 说 明
BackColorChanged	当 BackColor 属性的值更改时发生
BackgroundImageChanged	当 BackgroundImage 属性的值更改时发生
EnabledChanged	在 Enabled 属性值更改后发生
Click	在单击控件时发生
DoubleClick	在双击控件时发生
DragDrop	在完成拖放操作时发生
Move	在移动控件时发生
Enter	进入控件时发生
Leave	在输入焦点离开控件时发生
GotFocus	在控件接收焦点时发生
LostFocus	当控件失去焦点时发生
KeyDown	在控件有焦点的情况下按下键时发生
KeyPress	在控件有焦点的情况下按下键时发生
KeyUp	在控件有焦点的情况下释放键时发生
MouseClick	在鼠标单击该控件时发生
MouseDoubleClick	当用鼠标双击控件时发生
MouseDown	当鼠标指针位于控件上并按下鼠标键时发生
MouseEnter	在鼠标指针进入控件时发生
MouseHover	在鼠标指针停放在控件上时发生

续表

事件名称	事件说明
MouseLeave	在鼠标指针离开控件时发生
MouseMove	在鼠标指针移到控件上时发生
MouseUp	在鼠标指针在控件上并释放鼠标键时发生
MouseWheel	在移动鼠标轮并且控件有焦点时发生
TextChanged	在 Text 属性值更改时发生
VisibleChanged	在 Visible 属性值更改时发生
Resize	在调整控件大小时发生
RegionChanged	当 Region 属性的值更改时发生
SizeChanged	在 Size 属性值更改时发生

5.3.2　用 Label 显示静态文本

在.NET 类库中，Label 控件是常用的显示静态文本控件，它不能具有焦点，所以不能接收用户输入信息。但是通过它的属性和方法可以设置字体、颜色、背景色、背景图片等，使其具有丰富的视觉效果。Label 控件由类 System.Windows.Forms.Label 提供，同样是继承自 Control 类，除了 BackColor、ForeColor 等属性外，还包括表 5-8 所示的特有属性，如 Image、TextAlign 等。

表 5-8　Label控件特有成员

属性名称	属性说明
Text	Label 控件要显示的文本，通过 TextAlign 属性设置文本对齐方式
TextAlign	Label 控件中文本的对齐方式
Image	Label 控件的背景图片，可以通过 ImageAlign 属性设置图片对齐方式
ImageAlign	Label 控件的背景图片的对齐方式
AutoSize	Label 控件尺寸是否根据文本自动调整，ture 表示自动调整

示例代码 5-5 是实例 LabelControl 中由窗体设计器自动生成的代码片段，有两个 Label 控件 label1、label2，分别对它们设置不同的字体、颜色、背景色、边框等。

示例代码 5-5

```
//Label1 参数设置
this.label1.AutoSize = true;                              //自动调整大小
this.label1.BackColor = System.Drawing.Color.Red;         //背景色为红色
this.label1.Font = new System.Drawing.Font("楷体_GB2312",  //楷体
                                  12F,System.Drawing.FontStyle.Regular,
                                  System.Drawing.GraphicsUnit.Point,
                                  ((byte) (134)));
this.label1.ForeColor = System.Drawing.Color.White;       //前景色为白色
this.label1.Location = new System.Drawing.Point(67, 45);  //位置
this.label1.Name = "label1";                              //名称
this.label1.Size = new System.Drawing.Size(232, 16);      //大小
this.label1.TabIndex = 0;                                 //Tab 序号
this.label1.Text = "红底白字,楷体 小四号, 无边框";              //显示的文本
//label2 参数设置
this.label2.AutoSize = true;                              //自动调整大小
```

```
this.label2.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
                                                        //Single 边框
this.label2.Font = new System.Drawing.Font("幼圆",       //幼圆
                                  15F,
                                  System.Drawing.FontStyle.Bold,
                              System.Drawing.GraphicsUnit.Point,
                              ((byte) (134)));
this.label2.Location = new System.Drawing.Point(43, 82);  //位置
this.label2.Name = "label2";                              //名称
this.label2.Size = new System.Drawing.Size(289, 22);      //大小
this.label2.TabIndex = 1;                                 //Tab 序号
this.label2.Text = "幼圆 小三 粗体, Single 边框";          //显示的文本
```

示例代码 5-5 中创建的两个 Label 控件的效果如图 5-7 所示。Label1 为红底白字，楷体，小四号大小，并且无边框。而 Label2 为幼圆，小三，粗体，FixedSingle 边框颜色为默认的灰底黑字。

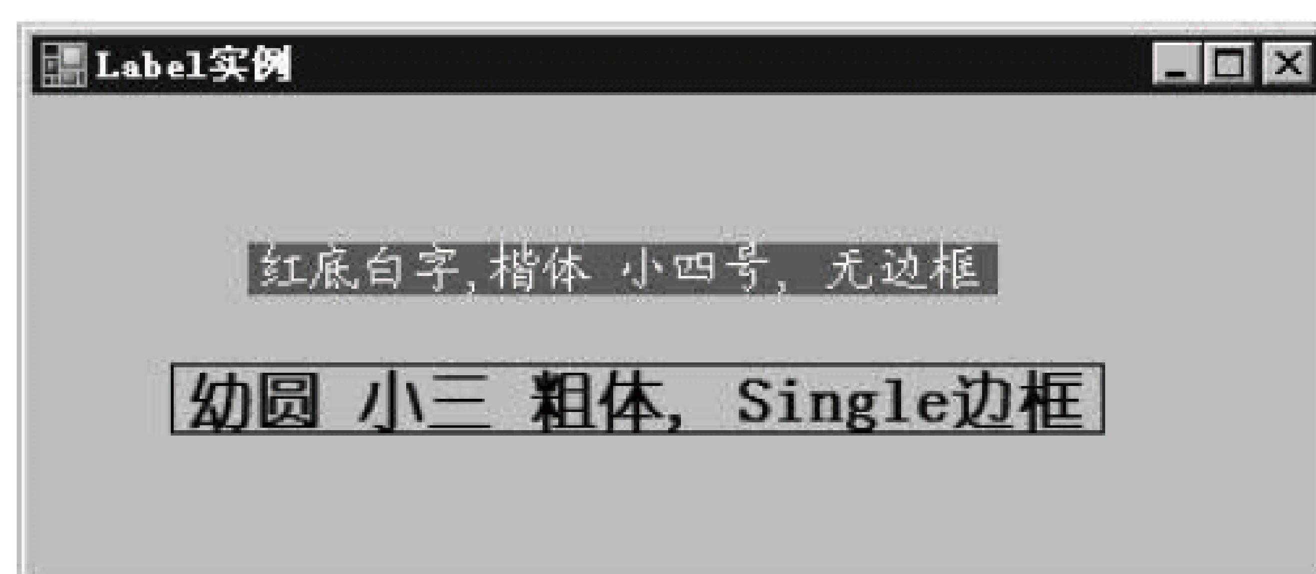


图 5-7　LabelControl 效果图

技巧： 实际开发中，可以将文本和图片结合，合理使用 ImageAlign、TextAlign 等属性搭配出图文并茂的 Label 界面。另外，Label 控件的文本等在程序运行期间只能通过代码进行修改。

5.3.3　用 Button 实现按钮

Button（按钮）是 Windows 应用程序中最常见的控件之一，可以说是随处可见。在.NET 类库中，Button 控件由 System.Windows.Forms.Button 类提供，除了显示文本之外，它只有一个重要事件——Click，即按钮单击事件。

从外观而言，Button 控件同样可以显示文本的图片，而且可以两者同时显示，通过 Text 和 TextAlign 属性设置其文本及对齐方式，通过 Image 和 ImageAlign 属性设置图片及对齐方式。另外，当同时显示图片和文字时可以通过 TextImageRelation 属性指定文字相对图片的位置，包括重叠、文字在图片上、图片在文字上、文字在图片前、图片在文字前 5 个选项。可以见 Button 控件也是可以相当美观的。

而外观不是 Button 控件的关键，它的关键在于 Click 事件。在一个 Windows 应用程序中，很多操作和逻辑运算都是由 Button 控件触发的，Button 的文本和图片通常为用户提供指引信息，用户根据需要单击 Button 后，引发 Button 的 Click 事件。开发人员只是根据应用程序的需要，在 Click 事件处理函数中添加适当的处理代码即可。关于如何添加控件事件响应函数，见 5.1.4 节。

示例代码 5-6 是实例 ButtonControl 中 Form1 类的主要代码，该窗体上包含 3 个 Button 控件和 1 个 Label 控件。如图 5-8 所示，Button 控件 btnAdd 的名称为“多一点”，btnSub 的名称为“少一点”，btnDeal 的名称为“成交”，而 Label 控件 lbRes 则用来显示当前的信息，这其实是模拟一个简单的在商店买东西的过程。其中 Form1 的_Value 字段表示当前花费，而 btnAdd_Click()和 btnSub_Click()分别表示增加和减少当前的花费。btnDeal_Click()显示出当前最终的花费。

示例代码 5-6

```
public partial class Form1 : Form
{
    private int _Value = 0;              //记录当前的值
    //btnShowMsg 的 Click 事件处理函数
    private void btnShowMsg_Click(object sender, EventArgs e)
    {
        MessageBox.Show(string.Format("现在是: {0}", this._Value));
    }
    //“多一点”按钮 Click 事件处理函数
    private void btnAdd_Click(object sender, EventArgs e)
    {
        this._Value++;
        this.lbRes.Text = string.Format("现在是: {0}", this._Value);
    }
    //“少一点”按钮 Click 事件处理函数
    private void btnSub_Click(object sender, EventArgs e)
    {
        this._Value--;
        this.lbRes.Text = string.Format("现
在是: {0}", this._Value);
    }
}
```

实例 ButtonControl 的效果如图 5-8 所示，从中可以看出，Button 控件本身也可以非常美观，而且结合它的 Click 事件，可以实现任何复杂的功能。



图 5-8　ButtonControl 效果图

5.3.4　用 CheckBox 和 RadioButton 实现选中

在实际开发中，很多时候需要在多个候选项中选择一个或多个，这就需要用到 CheckBox 控件和 RadioButton 控件。CheckBox 控件由 System.Windows.Forms.CheckBox 类提供，可以显示文本和图片给出提示信息，同时提供一个方框进行选中，选中时方框会被勾上，可以通过 Checked 属性获取或设置该选项的选中状态，多个 CheckBox 相互独立，也就是说它们可以同时被选中多个。

与 CheckBox 控件类似，RadioButton 控件也用于选中多个候选项中的一项。但是 RadioButton 控件由 System.Windows.Forms.RadioButton 类提供，而且多个 RadioButton 被作为一组相互排斥的选项，同时只能有一个被选中。同样可以通过 Checked 属性获取或设置选项的选中状态，当一个 RadioButton 被选中，同一组的其他 RadioButton 会自动取消选中。

另外，当 CheckBox 和 RadioButton 的选中状态发生变化时，都会引发 CheckedChanged 事件，通过处理该事件，可以监视到选中项的改变，然后做出相应的处理。示例代码 5-7 是实例 CheckRadioButtons 的主要代码，通过设计器添加 7 个 CheckBox 到 GroupBox “选择星期”中，分别表示星期一到星期天 7 个可以多选的候选项。添加 3 个 RadioButton 到 GroupBox “选择项目”中，分别表示 3 个相互排斥的候选项。添加 3 个 RadioButton 到 GroupBox “选择奖牌”中，分别表示另外 3 个相互排斥的候选项。

示例代码 5-7

```
public partial class Form1 : Form
{
    public Form1( )                       //窗体构造函数，绘制窗体
    {
        InitializeComponent( );
    }
    //所有 CheckBox 和 RadioButton 控件 CheckedChanged 事件处理函数，产生对应的文本
    private void buttons CheckedChanged(object sender, EventArgs e)
    {
        string weeks = "";                //根据 7 个候选项产生多个选中的星期信息
        weeks += ckbWeek1.Checked ? "星期一 " : "";
        weeks += ckbWeek2.Checked ? "星期二 " : "";
        weeks += ckbWeek3.Checked ? "星期三 " : "";
        weeks += ckbWeek4.Checked ? "星期四 " : "";
        weeks += ckbWeek5.Checked ? "星期五 " : "";
        weeks += ckbWeek6.Checked ? "星期六 " : "";
        weeks += ckbWeek7.Checked ? "星期日 " : "";
        string sport = "";                //根据第一组 RadioButton 产生唯一的体育项目
        sport += rbSword.Checked ? "击剑 " : "";
        sport += rbJump.Checked ? "跳水 " : "";
        sport += rbTiCao.Checked ? "体操 " : "";
        string jiangpai = "";             //根据第二组 RadioButton 产生唯一的奖牌
        jiangpai += rbGold.Checked ? "金牌 " : "";
        jiangpai += rbSliver.Checked ? "银牌 " : "";
        jiangpai += rbTong.Checked ? "铜牌 " : "";
        this.lbHint.Text = weeks + sport + jiangpai;
    }

}
```

如图 5-9 所示，是实例 CheckRadioButtons 的运行效果，其中各个按钮的颜色是在窗体设计器中设计的，并非代码产生。最下面 Label 控件 lbHint 的文本是在 CheckBox 和 RadioButton 的选中状态发生变化时动态产生。而且，由于“选择项目”和“选择奖牌”属于两个不同的分组，所以它们可以各选一项。

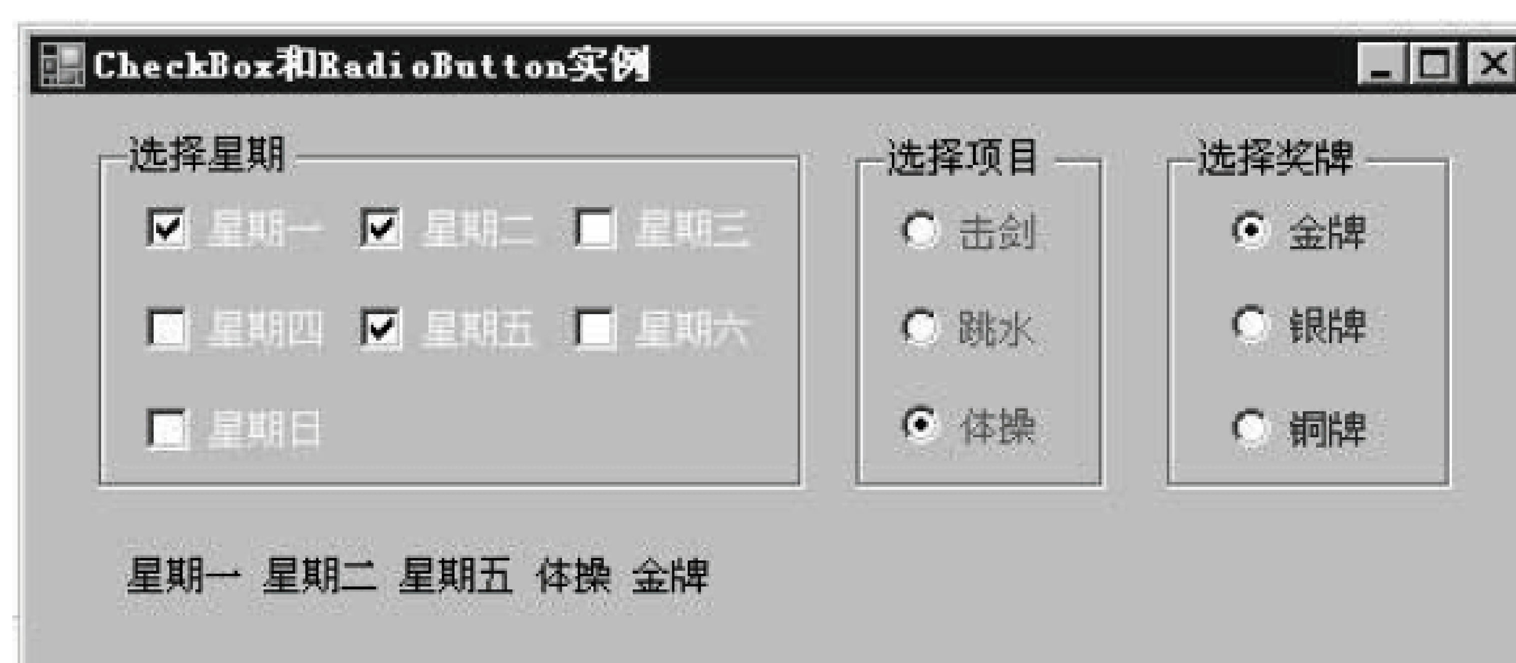


图 5-9　CheckBox 和 RadioButton 效果图

5.3.5　用 TextBox 和 MaskedTextBox 输入文本

在.NET 类库中，TextBox 控件是最常用和最简单的文本显示和输入控件，它可以设置字体、前景色、背景色等，还通过右键菜单提供复制、粘贴、剪切等常用文本操作，但是

它不能设置背景图片。TextBox 控件是由 System.Windows.Forms.TextBox 类提供的，表 5-9 为它的常用成员。

表 5-9　TextBox控件常用成员

名　称	说　明
BackColor	TextBox 控件的背景色
ForeColor	TextBox 控件文本的颜色
Font	TextBox 控件文本的字体，字体、大小、粗体、斜体、删除线等
Text	TextBox 控件要显示的文本，通过 TextAlign 属性设置文本对齐方式
PasswordChar	TextBox 控件以密码输入方式使用，输入字符用该属性指定字符屏蔽
Enable	TextBox 控件是否可用，不可用，则灰色显示，不能使用右键菜单
Visible	TextBox 控件是否可见，不可见，则隐藏
ReadOnly	TextBox 控件是否只读，可以使用右键菜单复制操作
MultiLine	TextBox 控件是否包含多行文本
Clear	清除 TextBox 控件中已有的文本
AppendText	在 TextBox 控件最后追加文本
TextChanged	TextBox 控件中文本发生变化时引发的事件，每次输入字符都引发一次
Enter	TextBox 控件获得焦点时引发的事件
Leave	TextBox 控件失去焦点时引发的事件

在一些特殊数据的输入时，数据的格式有限制，如整数、YYYY-MM-DD 格式的日期等。可以通过由 System.Windows.Forms.MaskedTextBox 类提供的 MaskedTextBox 控件来完成特定格式数据的输入，它的常用成员与 TextBox 相同，但它不能多行输入。MaskedTextBox 包括一个 string 类型的 Mask 属性，用来描述合法数据的格式，通过“属性编辑器”编辑 MaskedTextBox 的 Mask 属性，从“输入掩码”窗体中选择常用的掩码格式，它们由 Visual Studio 2010 提供，如图 5-10 所示。



图 5-10　常用掩码列表选择窗体

另外，Mask 格式字符串还可以根据需要自定义，它必须是由一个或多个掩码元素组成的字符串。掩码元素见表 5-10，比如要输入一个最多 10 位最少 3 位的整数，它的掩码为

“9999999000”，即低 3 位数字且必需，高 7 位数字可选。

表 5-10　MaskedTextBox控件掩码元素

掩 码 元 素	说　　明
0	接受 0～9 之间的任何一个数字，必须输入
9	数字或空格，可选输入
#	数字或空格，且允许使用加号“+”和减号“-”，可选输入
L	ASCII 字母 a～z 和 A～Z，必需输入
?	ASCII 字母 a～z 和 A～Z，可选输入
&	任何字符，必须输入
C	任何非控制字符，可选输入
A	0～9 之间的数字，ASCII 字母 a～z 和 A～Z，可选输入
a	0～9 之间的数字，ASCII 字母 a～z 和 A～Z，可选输入
.	小数点占位符
,	千分位占位符
:	时间分隔符
/	日期分隔符
$	货币符号
<	后续所有字符都转换为小写
>	将后续所有字符都转换为大写
|	禁用前一个大写转换或小写转换
\	对掩码字符进行转义，将其转变为原义字符，如“\\”是反斜杠的转义序列
其他所有字符	原义字符，在运行时始终占据掩码中的一个固定位置，不能移动或删除该字符

示例代码 5-8 是实例 TextBoxControl 的具体实现，窗体中包括两个 TextBox 控件，tbName 是单行文本框用于输入用户姓名，tbUsers 是多行文本框用于显示用户信息；包括两个 MaskedTextBox 控件，mtbMobile 的 Mask 属性为“13000000000”只能接收手机号码，mtbPhone 的 Mask 属性为“0000-90000000”只能接收电话号码。

示例代码 5-8

```
public partial class Form1 : Form
{
    //窗体加载时设置 mtbMobile 和 mtbPhone 的 Mask 值
    private void Form1_Load(object sender, EventArgs e)
    {
        this.mtbMobile.Mask = "13000000000"; //手机号码：13 后面 9 个必填数字
        this.mtbPhone.Mask = "0000-90000000";//电话：4 位必填区号，7 或 8 位号码
        this.tbName.Text = "";
        this.tbUsers.Text = "";
    }
    //添加按钮将输入的用户信息添加到 tbUsers 中，并清空输入文本框
    private void btnAdd_Click(object sender, EventArgs e)
    {
        string usr = string.Format("<{0}>:<{1}>:<{2}>", //产生用户信息
                                   this.tbName.Text,
                                   this.mtbPhone.Text,
                                   this.mtbMobile.Text);
        //添加到用户记录文本框
        this.tbUsers.AppendText(usr + System.Environment.NewLine);
```

```
        this.mtbMobile.Text = "";                       //清空用户信息
        this.mtbPhone.Text = "";
        this.tbName.Text = "";
    }
}
```

如图 5-11 所示为 TextBoxControl 类的效果图，从中可以看出 MaskTextBox 控件需要输入的位置用下划线“_”作为占位符，并且只能输入规定的字符。



图 5-11　TextBoxControl 实例效果图

5.3.6　用 ListBox 和 ComboBox 实现选中

列表也是 Windows 界面中的常见控件，主要有 ComboBox（下拉框）和 ListBox（列表）两个。.NET 类库中，下拉框控件由 System.Windows.Forms.ComboBox 类提供，它以常见的下拉框方式给出所有可选项，可选项也可以通过代码动态变化。根据用户操作确定具体的选择项和文本值，同时还可以手动输入要选择的值，比 RadioButton 方便灵活。表 5-11 列出了 ComboBox 的主要成员。

表 5-11　ComboBox控件常用成员

名　称	说　明
BackColor	ComboBox 控件的背景色
ForeColor	ComboBox 控件文本的颜色
Font	ComboBox 控件文本的字体，字体、大小、粗体、斜体、删除线等
Text	ComboBox 控件要显示的文本，通过 TextAlign 属性设置文本对齐方式
Enable	ComboBox 控件是否可用，不可用，则灰色显示，不能使用右键菜单
Visible	ComboBox 控件是否可见，不可见，则隐藏
DropDownStyle	ComboBox 控件下拉框选择样式，可选 Simple、DropDown、DropDownList
Sorted	ComboBox 控件中的可选项是否进行排序
Items	ComboBox 控件中可选项列表
SelectedIndex	ComboBox 控件中被选中项的索引，从 0 开始
SelectedItem	ComboBox 控件中被选中的项
SelectedIndexChanged	ComboBox 控件中被选中项的索引发生变化时引发的事件

ComboBox 控件中下拉框样式由 DropDownStyle 属性设置，该属性具有 3 个可选项。

- Simple：没有下拉框，所以不能选择，可以输入，和 TextBox 控件相似。
- DropDown：具有下拉框，可以选择，也可以直接输入选择项中不存在的文本。
- DropDownList：具有下拉框，只能选择下拉框中已有的候选项，不能输入其他文本。

和 ComboBox 控件相似，ListBox 也提供对列表中元素的选择，但是它支持多选，而且以列表的样式给出所有的候选项。下拉框控件由 System.Windows.Forms.ListBox 类提供，ListBox 控件的选择模式由 SelectionMode 属性设置，该属性具有 4 个可选项。

- None：不能选中任意可选项。
- One：同时只能选中一个可选项。
- MultiSimple：可以多选，但每次鼠标操作只能选中或取消选中一个可选项，可以配合 Ctrl 和 Shift 键完成跳跃或连续选中。
- MultiExtended：可以多选，且可以通过鼠标拖放来一次选中多个可选项，还可以配合 Ctrl 和 Shift 键完成跳跃或连续选中。

ComboBox 和 ListBox 控件中可选项都通过 Items 进行操作，比如增加、删除、修改等，所以它们并没有提供直接修改候选项的接口。

示例代码 5-9 是实例 ComboListBox 的主要代码，窗体上包括一个 ComboBox 控件 cmbHouXuan，它的可选值是随机产生的，一个 ListBox 控件 lstResults，它的可选值是从 cmbHouXuan 中选出来的。在窗体加载时设置 cmbHouXuan 只能从已有值中选择，设置 lstResults 只能单选且对所有项排序。

示例代码 5-9

```
public partial class Form1 : Form
{
    private void Form1 Load(object sender, EventArgs e)
    {
        //设置 cmbHouXuan 只能从 ComboBox 中的已有候选值中选择
        this.cmbHouXuan.DropDownStyle = ComboBoxStyle.DropDownList;
        //lstResult 只能执行单选，并且对所有值进行排序
        this.lstResults.SelectionMode = SelectionMode.One;
        this.lstResults.Sorted = true;
        this.GenerateCombItems( );          //产生 ComboBox 中的可选项
    }
    //随机产生 10 条新的候选值到 CombBox 控件中
    private void GenerateCombItems( )
    {
        this.cmbHouXuan.Items.Clear( );     //移除原有的数据
        Random rd = new Random();
        for (int i = 0; i < 10; i++)        //随机生成 10 个新的数据
        {
            string item = string.Format("Item-{0:X8}", rd.Next( ));
            this.cmbHouXuan.Items.Add(item); //添加到 ComboBox 中
        }
        this.cmbHouXuan.SelectedIndex = 0;  //默认选中第一条
    }
    //重新生成 ComboBox 中的候选项
    private void btnFresh Click(object sender, EventArgs e)
    {
        this.GenerateCombItems( );          //重新生成 CombBox 中的候选项
    }
    //将 CombBox 中选中的值添加到 ListBox 中
    private void btnAddOne Click(object sender, EventArgs e)
    {
        //通过 ComboBox.SelectedItem 获取当前选中的候选项，然后添加到 ListBox 中
        string item = (string)this.cmbHouXuan.SelectedItem;
```

```
        this.lstResults.Items.Add(item);
    }
    //从 ListBox 中移除当前选中项
    private void btnRemoveOne_Click(object sender, EventArgs e)
    {
        if (this.lstResults.SelectedIndex >= 0) //如果当前 ListBox 中有选中条
                                                 目，则移除它
        {
this.lstResults.Items.RemoveAt(this.lstResults.SelectedIndex);
        }
    }
    //从 ListBox 中移除所有项
    private void btnRemovAll_Click(object sender, EventArgs e)
    {
        this.lstResults.Items.Clear( );          //移除所有项
    }
}
```

如图 5-12 是示例 ComboListBox 的运行效果图，从中可以看出，ListBox 中的所有项都是从小到大进行排序的，而且 ComboBox 中的确也不能输入，只能选择已有项。



图 5-12　ComboListBox 实例效果图

注意：ListBox 和 ComboBox 的 Items 属性都是 Object 类型集合，所以它们可以包含任意的数据类型，甚至每个可选项的数据类型都可以各不相同，界面上显示该对象 ToString 方法得到的字符串。

5.3.7　用 TabControl 实现动态分组

GroupBox（分组框）、Pannel（面板）和 TabControl（选项卡）是常用的 3 种容器类控件，它们可以包含其他控件，其中前面两个都是静态的，而且使用非常简单，本书不做介绍。.NET 类库提供了选项卡控件由 System.Windows.Forms.TabControl 类提供，它在参数配置界面中经常遇到，可以包含 0 或多个具有标题的选项页（TabPage）。选项页由 System.Windows.Forms.TabPage 类实现，类似一个 Pannel，可以包含不同的控件，完成不

同的功能。选项卡中的选项页还可以通过代码动态增加、删除、隐藏等。如表 5-12 所示为 TabControl 控件常用成负。

表 5-12　TabControl控件常用成员

名　称	说　明
MultiLine	TabControl 控件中选项卡太多时，是否采用多行排列样式
Appearance	TabControl 控件选项卡标签的样式，可选 Buttons、FlatButtons、Normal
Aligment	TabControl 控件选项卡标签的排列方向，可选 Top、Bottom、Left、Right
Enable	TabControl 控件是否可用，不可用，则灰色显示，不能使用右键菜单
Visible	TabControl 控件是否可见，不可见，则隐藏
TabPages	TabControl 控件所包含的选项页 TabPage 集合

TabControl 控件的外观如图 5-13 所示，这里包含 2 个 TabPage 控件，第 1 个名为“设置颜色”，第 2 个名为“设置字体”。TabControl 控件通过 TabPages 属性来管理它所包含的选项卡，通过 TabPages 的 Add()、Remove()等方法来添加、删除选项页，由于不太常用，鉴于篇幅关系本书不再深入介绍。

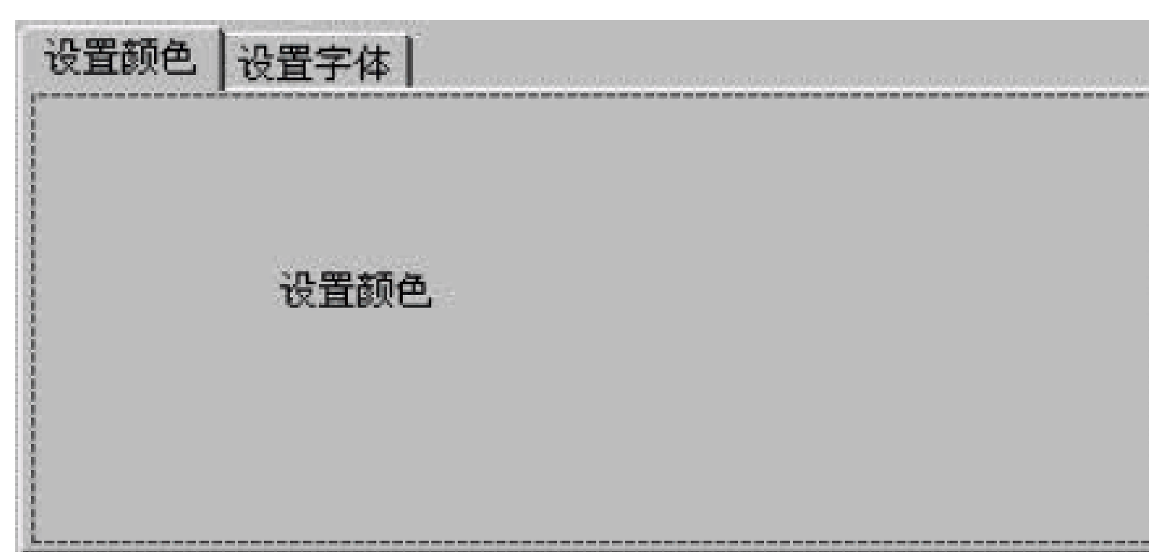


图 5-13　TabControl 效果图

5.4　使用菜单和工具栏

一个友好的用户界面一定离不开菜单和工具栏，菜单和工具栏通常提供一些快捷方式来使用软件，有时需要通过状态栏显示软件的最新状态信息。.NET 类库同样提供了这些控件，可以方便地在窗体中增加菜单、工具栏和状态栏。

5.4.1　用 MenuStrip 和 ContentMenuStrip 实现菜单

菜单（Menu）通常分为两类，即主菜单和上下文菜单（又叫快捷菜单），在.NET 类库中分别提供了 MenuStrip 和 ContentMenuStrip 控件来实现主菜单和上下文菜单。

MenuStrip 控件由 System.Windows.Forms.MenuStrip 类实现，它必须依附在某个窗体上作为该窗体的主菜单，通常显示在窗体的最上方。通常包含多个不同的菜单项（MenuItem），并且可以通过代码动态的添加或删除菜单项。MenuStrip 可以包含 4 种不同类型的菜单项。

- ❑ MenuItem 类型：类似 Button 的菜单项，通过单击实现某种功能，同时可以包含子菜单项，它以右三角形的形式表示包含子菜单。
- ❑ ComboBox 类型：类似 ComboBox 控件的菜单项，可以在菜单中实现多个可选项的选择。
- ❑ TextBox 类型：类似 TextBox 控件的菜单项，可以在菜单中输入任意文本。
- ❑ Separator 类型：菜单项分隔符，以灰色的“—”表示。

菜单项作为一种特殊的控件，同样可以通过 BackColor、ForeColor、Font 等属性来设置显示外观，使它更具特色。不同类型菜单项具有不同的常用事件需要处理，MenuItem 类

型通常处理 Click 事件来完成单击当前菜单需要执行的操作。ComboBox 类型则通常处理 SelectedIndexChanged 事件来判断选择变动的处理，同时也可以提供用户数据的输入和输出。TextBox 类型则主要是提供用户数据的输入，也可以响应 TextChanged、KeyPress 等事件实现一些扩展功能。

在 Visual Studio 2010 中，只需要从“工具箱”中将 MenuStrip 控件拖放到目标窗体上，就为该窗体添加了主菜单。然后通过窗体设计器或项集合编辑器（如图 5-14 所示）以树形结构的方式编辑任何菜单项及其子菜单项。如果需要设置菜单为其他窗体的主菜单，则手动将窗体的 MainMenuStrip 设置为该菜单即可。

上下文菜单（快捷菜单）的设计和编辑都与主菜单完全相同，只是它被作为右键弹出菜单使用，任何控件都具有一个 ContextMenuStrip 属性，用来表示在控件上发生鼠标右击事件时要弹出的快捷菜单。值得注意的是，MainMenuStrip 通常不需要手动设置，而 ContextMenuStrip 则需要手动或代码来设置。

图 5-15 是实例 MenuStrip 中弹出菜单的效果图，其中窗体顶部的“文件”和“帮助”是主菜单，弹出菜单“复制”等在文本框上右击即产生，因为将它的 ContextMenuStrip 设置为菜单“复制”，关于该实例的更多细节见本书光盘。

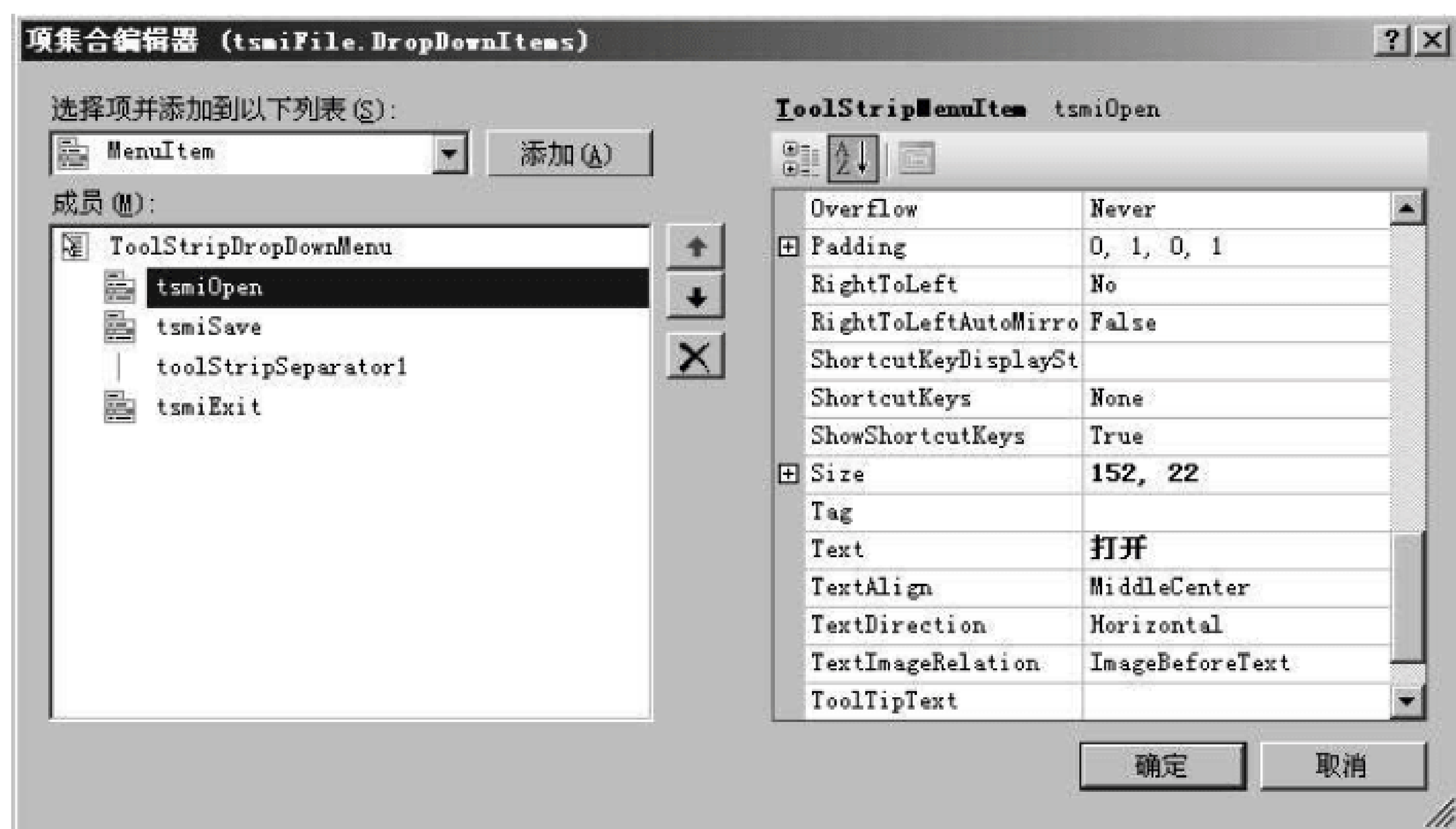


图 5-14　菜单编辑器界面



图 5-15　MenuStrip 实例运行效果

5.4.2　用 ToolStrip 控件实现工具栏

工具栏是另一种快速执行的常用方式之一，.NET 类库中提供了 ToolStrip 控件实现工具栏界面。工具栏必须停靠在某个窗体或控件上，可以包含多个工具栏项（ToolStripItem），不同的项具有不同的功能和意义。在.NET 类库中，工具栏可以包含 8 种不同类型的工具栏项。

- ❑ Label 类型：和 Label 控件类似，常在工具栏上提示静态文本。
- ❑ Button 类型：和 Button 控件类似，通常通过鼠标单击（Click）事件来引发某个具体的操作。
- ❑ ComboBox 类型：和 ComboBox 控件类似，通常在工具栏提供一些可选项的选择，并通过 SelectedIndexChanged 事件引发某个具体的操作。
- ❑ TextBox 类型：和 TextBox 控件类似，通常在工具栏让用户输入数据。
- ❑ SplitButton 类型：具有下拉菜单的工具栏项目。

- ❑ DropDownButton 类型：具有下拉菜单的工具栏项目。
- ❑ ProgressBar 类型：进度条样式工具栏项目，通常在工具栏进行进度提示。
- ❑ Separator 类型：工具栏分隔符，以灰色的“|”表示。

在 Visual Studio 2010 中，添加工具栏只需要从“工具箱”将 ToolStrip 控件拖放到目标窗体或控件上即可，然后通过窗体设计器或项集合编辑器（如图 5-14 所示）以树形结构的方式编辑任何工具栏项目及其子项。

工具栏项也是控件，同样可以通过 Image、BackColor、ForeColor、Font 等属性来设置显示外观，使它更具特色。工具栏最主要的开发工作在于为每个工具栏项根据软件需要添加事件处理函数，从而实现不同的功能。不同类型的工具栏项具有不同的事件需要处理，Button 类型通常处理 Click 事件完成单击当前工具栏项需要执行的操作。ComboBox 类型则通常处理 SelectedIndexChanged 事件来判断选择变动的处理，同时也可以提供用户数据的输入和输出。TextBox 类型则主要是提供用户数据的输入，也可以响应 TextChanged、KeyPress 等事件实现一些扩展功能。ProgressBar 类型则主要用于显示，并不需要处理任何事件，但是要实时更新。

如图 5-16 所示为实例 MenuStrip，在菜单的基础上增加工具栏之后的运行效果，按照习惯将主菜单窗体的顶部，将工具栏放在菜单下一行。



图 5-16　工具栏运行效果

5.4.3　用 StatusStrip 控件实现状态栏

与 MenuStrip 和 ToolStrip 正好相反，StatusStrip（状态栏）常在窗体的最下方，以只读的方式提示当前软件的运行状态等信息，.NET 类库提供 StatusStrip 控件方便地实现状态栏界面。状态栏必须停靠在某个窗体或控件上，可以包含多个状态项（StatusStripItem），不同的项具有不同的功能和意义。在.NET 类库中，状态栏可以包含 4 种不同类型的状态栏项目。

- ❑ Label 类型：和 Label 控件类似，常在状态栏上提示静态文本。
- ❑ SplitButton 类型：具有下拉菜单的状态栏项目。
- ❑ DropDownButton 类型：具有下拉菜单的状态栏项目。
- ❑ ProgressBar 类型：进度条样式状态栏项目，通常在状态栏进行进度提示。

状态栏项同样是从 Control 类继承而来的控件，可以通过 Img、BackColor、ForeColor、Font 等属性来设置显示外观，使它更具特色。

要添加状态栏，只需要从“工具箱”将 StatusStrip 控件拖放到目标窗体或控件上即可。然后通过窗体设计器或项集合编辑器（如图 5-14 所示）以树形结构的方式编辑任何工具栏项目及其子项。由于状态栏的数据通常表示软件的最新状态，所以它往往为只读，而且要注意状态的实时性和正确性，同时要注意清除不必要的状态，从而提高界面的友好性。

如图 5-17 所示为 MenuStrip 实例增加了状态栏之后的运



图 5-17　状态栏运行效果

行效果图，从中可以看出，整个窗体集成了菜单、工具栏、状态之后，看起来更加友好和专业，给人耳目一新的感觉。

5.5　使用通用对话框

通用对话框是指 Windows 下常用的具有统一风格的对话框，.NET 类库提供多种通用对话框，主要包括消息对话框、打开文件对话框、保存文件对话框、颜色选择对话框、字体选择对话框等。本节将介绍这些对话框的具体使用。

5.5.1　用 MessageBox 显示提示消息

在 Windows 中，MessageBox（消息对话框）是最常用和简单的提供文本消息的方式，它以弹出式对话框的方式将要提示的文本信息呈现给用户，包括标题、提示信息、按钮类型等属性可以设置。在.NET 类库中，通过 System.Windows.Forms.MessageBox 类来实现消息对话框。

MessageBox 类最主要的一个成员是静态方法 Show()，该成员用来以阻塞的方式显示一个 Windows 消息框，它包含多达 21 个不同的重载，可以实现多种样式的对话框。其中最常用的是以下 4 个：

- MessageBox.Show(string text)：显示一个提示内容为 text，带“确定”按钮的消息框，返回 DailogResult.OK，即确定按钮。
- MessageBox.Show(string text, string caption)：显示一个提示内容为 text，标题为 caption，带“确定”按钮的消息框，返回 DailogResult.OK，即确定按钮。
- MessageBox.Show(string text, string caption, MessageBoxButtons buttons)：显示一个提示内容为 text、标题为 caption、带有 buttons 指定按钮的消息框，返回用户单击按钮的值。
- MessageBox.Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon)：显示一个提示内容为 text、标题为 caption、提示图标为 icon，带有 buttons 指定按钮的消息框，返回用户单击按钮的值。

其中，MessageBox.Show()方法的参数之一 buttons 为 MessageBoxButtons 枚举，用来表示要显示的按钮，包含以下几个可选值：

- MessageBoxButtons.OK：只有确定按钮。
- MessageBoxButtons.OKCancel：包括确定和取消按钮。
- MessageBoxButtons.AbortRetryIgnore：包括终止、重试和忽略按钮。
- MessageBoxButtons.YesNoCancel：包括是、否和取消按钮。
- MessageBoxButtons.YesNo：包括是和否按钮。
- MessageBoxButtons.RetryCancel：包括重试和取消按钮。

此外，MessageBox.Show()方法的返回类型为 DialogResult 枚举，包含以下几个可选值。

- DialogResult.None：没有按钮单击，直接通过关闭按钮关闭消息框。
- DialogResult.OK：单击“确定”按钮返回。

- DialogResult.Cancel：单击“取消”按钮返回。
- DialogResult.Abort：单击“终止”按钮返回。
- DialogResult.Retry：单击“重试”按钮返回。
- DialogResult.Ignore：单击“忽略”按钮返回。
- DialogResult.Yes：单击“是”按钮返回。
- DialogResult.No：单击“否”按钮返回。

示例代码 5-10 是实例 CommonDlgs 中按钮 btnMsgBox 的 Click 事件处理函数，它通过 MessageBox 依次显示 4 个不同类型的消息框。前 2 个只有确定按钮，如图 5-18 所示，第 3 个具有“确认”和“取消”两个按钮，如图 5-19 所示，第 4 个还包括一个告警图标，如图 5-20 所示。

示例代码 5-10

```
//依次显示 4 个不同类型的消息框
private void btnMsgBox_Click(object sender, EventArgs e)
{
    //显示最简单的 MessageBox
    MessageBox.Show("这是第一个消息框，只有确认按钮");
    //显示有文本和标题的 MessageBox
    MessageBox.Show("这是第二个消息框，有标题，只有确认按钮", "第二个消息框");
    //显示具有文本、标题、确定和取消按钮的 MessageBox
    MessageBox.Show("这是第三个消息框，有标题，只有确认和取消按钮",
                "第三个消息框", MessageBoxButtons.OKCancel);
    //显示具有文本、标题、确定和取消按钮、告警图标的 MessageBox
    MessageBox.Show("这是第四个消息框，有标题，只有确认和取消按钮，告警图标",
                "第四个消息框", MessageBoxButtons.OKCancel, MessageBoxIcon.
                Warning);
}
```



图 5-18　带“确定”按钮的 MessageBox

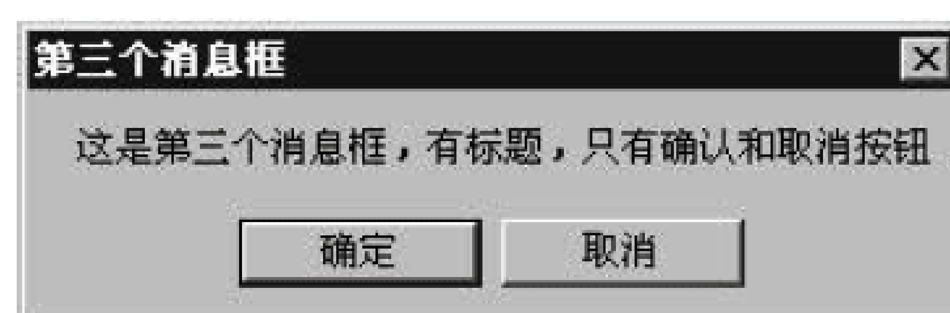


图 5-19　带 2 个按钮的 MessageBox

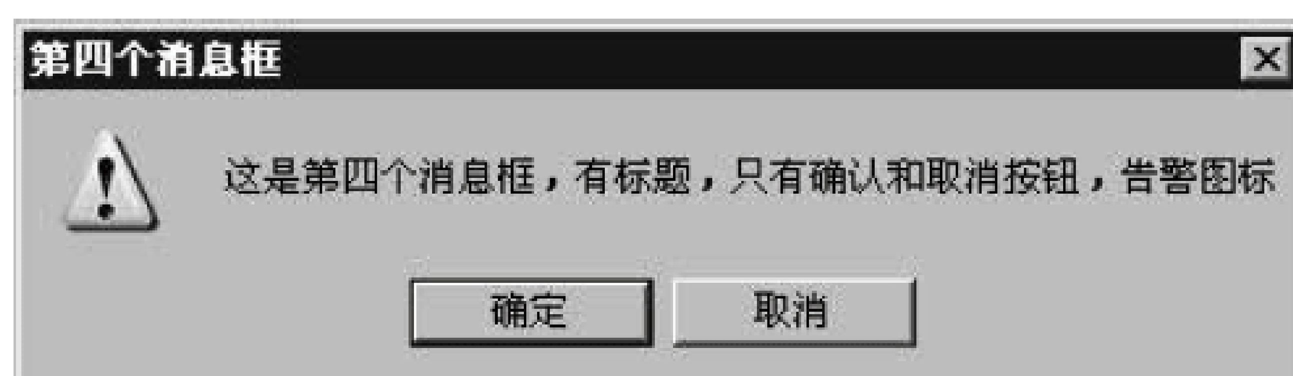


图 5-20　带告警图标的 MessageBox

5.5.2　用 OpenFileDialog 选择要打开的文件

在一些和文件操作有关的应用程序中，通常需要 Windows 打开文件对话框来浏览和选择文件。在.NET 类库中通过 System.Windows.Forms.OpenFileDialog 类来实现通用的打开文

件对话框，表 5-13 给出了它的主要成员。

表 5-13　OpenFileDialog常用成员

成 员 名 称	成 员 说 明
OpenFileDialog	构造函数，创建一个打开文件对话框对象
ShowDialog	Public 方法，以阻塞的方式显示打开文件对话框，并返回用户操作结果
Title	Public 属性，获取或设置显示在对话框上的标题
Filter	Public 属性，获取或设置扩展名的过滤信息，如“文本文件(*.txt)|*.TXT”表示只需要“*.TXT”文件，描述为“文本文件（*.txt）”。多个文件扩展名用竖线“|”分隔
Multiselect	Public 属性，获取或设置是否可以选择多个文件
FileName	Public 属性，获取打开文件对话框当前选中的文件名，包括路径和扩展名。或者设置默认选中的文件名
FileNames	Public 属性，获取当前选中的文件列表，Multiselect 为 true 时可以有多个元素

示例代码 5-11 是实例 CommonDlgs 中，按钮 btnOpenFile 的 Click 事件处理函数，首先创建一个 OpenFileDialog 对象 ofdlg，然后设置它的 Filter 属性，表示只选择扩展名为*.TXT 和*.DOC 的文件。然后通过 Title 属性设置对话框的标题，最后通过 ShowDialog()方法显示对话框。

示例代码 5-11

```
private void btnOpenFile Click(object sender, EventArgs e)
{
    OpenFileDialog ofdlg = new OpenFileDialog( );
                                            //创建 OpenFileDialog 对象
    ofdlg.Filter = "文本文件(*.txt)|*.TXT|Word 文件(*.doc)|*.DOC";
                                            //只选择 TXT 和 DOC 扩展名文件
    ofdlg.Title = "选择文本文件或 Word 文件";     //设置对话框的标题
    if(ofdlg.ShowDialog() == DialogResult.OK)   //显示对话框，并等待返回
    {
        this.tbOpenFileName.Text = ofdlg.FileName;  //如果用户选择了文件则显
                                                      示到界面
    }
    else
    {
        this.tbOpenFileName.Text = "还没有选择要打开的文件";
                                            //没有选择文件，则显示默认提示
    }
}
```

图 5-21 为示例代码 5-13 所产生的打开文件对话框，从中可以看出它和 Windows 系统自带的打开文件对话框一样，只是标题和文件过滤信息由代码产生。

5.5.3　用 SaveFileDialog 选择要保存的文件

和 OpenFileDialog 类相反，System.Windows.Forms.SaveFileDialog 类实现通用的保存文件对话框。表 5-14 给出了它的主要成员，可以看出它的成员和 OpenFileDialog 类似。



图 5-21　打开文件对话框效果图

表 5-14　SaveFileDialog常用成员

成 员 名 称	成 员 说 明
SaveFileDialog	构造函数，创建一个保存文件对话框对象
ShowDialog	Public 方法，以阻塞的方式显示打开文件对话框，并返回用户操作结果
Title	Public 属性，获取或设置显示在对话框上的标题
Filter	Public 属性，获取或设置新文件的扩展名，如“文本文件(*.txt)|*.TXT”表示扩展名为“*.TXT”。多个扩展名用竖线“|”分隔
FileName	Public 属性，获取保存文件对话框当前选中（或输入）的目标文件名，包括路径和扩展名。或者设置默认的目标文件名

示例代码 5-12 是实例 CommonDlgs 中按钮 btnSaveFile 的 Click 事件处理函数，首先创建一个 SaveFileDialog 对象 sfdlg，然后设置它的 Filter 属性，表示新文件扩展名为*.TXT。然后通过 Title 属性设置对话框的标题，最后通过 ShowDialog()方法显示对话框，用户可以选择或输入目标文件。

示例代码 5-12

```
private void btnSaveFile_Click(object sender, EventArgs e)
{
    SaveFileDialog sfdlg = new SaveFileDialog(); //创建 SaveFileDialog 对象
    sfdlg.Filter = "文本文件(*.txt)|*.TXT";          //默认扩展名为*.TXT
    sfdlg.Title = "请选择或输入要保存的文本文件";
    if(sfdlg.ShowDialog() == DialogResult.OK)       //显示对话框，并等待返回
    {
        this.tbSaveFileName.Text = sfdlg.FileName;  //如果用户选择了文件则显
                                                      示到界面
    }
    else
    {
        this.tbSaveFileName.Text = "还没有选择要保存的文件";
                                                 //没有选择文件，则显示默认提示
    }
}
```

5.5.4　用 ColorDialog 选择任意颜色

在很多应用软件中，比如图像处理、界面美化、文本编辑等，通常需要为窗体、编辑框等控件设置颜色，这就需要使用如图 5-22 所示的颜色选择对话框。System.Windows.Forms.ColorDialog 类实现 Windows 下标准的颜色选择对话框，表 5-15 给出了它的主要成员。

表 5-15　ColorDialog常用成员

成 员 名 称	成 员 说 明
ColorDialog	构造函数，创建一个颜色选择对话框对象
ShowDialog	Public 方法，以阻塞的方式显示打开文件对话框，并返回用户操作结果
Color	Public 属性，获取对话框中所选择的颜色，或设置对话框打开时选中的默认颜色

示例代码 5-13 是实例 CommonDlgs 中，按钮 btnSetColor 的 Click 事件处理函数，首先创建一个 ColorDialg 对象 cdlg，通过 Color 属性设置它的默认值为 btnSetColor 的前景色，然后通过 ShowDialog()显示对话框。如果用户选择了新的颜色，则将新颜色设置为 btnSetColor 的前景色。

示例代码 5-13

```
private void btnSetColor Click(object sender, EventArgs e)
{
    ColorDialog cdlg = new ColorDialog( );      //创建 ColorDialog 对象
    cdlg.Color = btnSetColor.ForeColor;
                                        //设置默认颜色为 btnSetColor 当前前景色
    if (cdlg.ShowDialog( ) == DialogResult.OK)  //显示对话框，并等待返回
    {
        this.btnSetColor.ForeColor = cdlg.Color;
                                        //选择了新的颜色，则更新 btnSetColor 前景色
    }
}
```

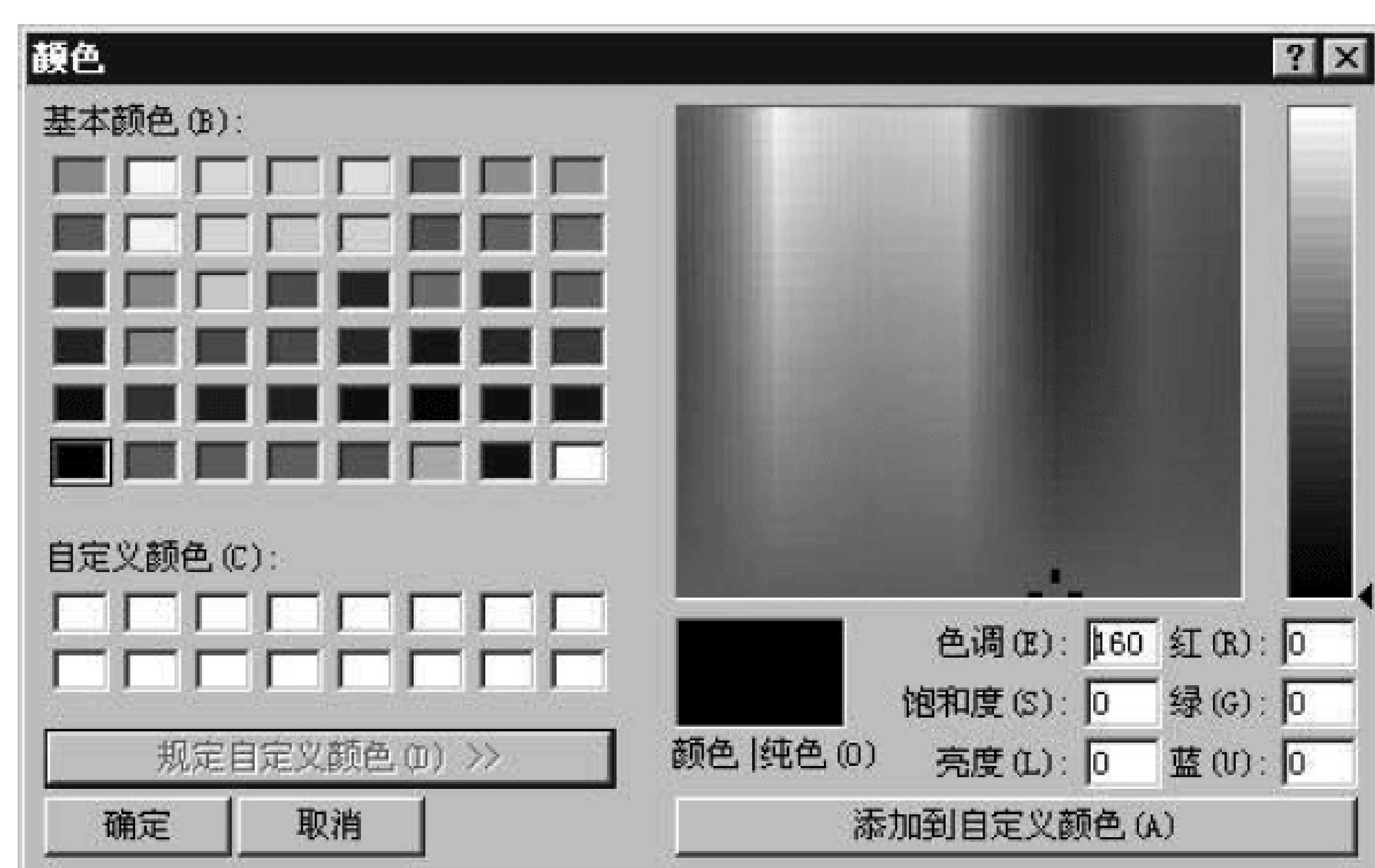


图 5-22　颜色选择对话框效果图

5.5.5　用 FontDialog 选择字体

和颜色选择一样，字体选择也是文本编辑等很多软件中常用的功能，在.NET 类库中，

通过 System.Windows.Forms.FontDialog 类实现如图 5-23 所示的字体选择对话框。表 5-16 给出了它的主要成员。

表 5-16　FontDialog常用成员

成 员 名 称	成 员 说 明
FontDialog	构造函数，创建一个字体选择对话框对象
ShowDialog	Public 方法，以阻塞的方式显示打开文件对话框，并返回用户操作结果
Font	Public 属性，获取对话框中所选择的字体，或设置对话框打开时选中的默认字体

示例代码 5-14 是实例 CommonDlgs 中，按钮 btnSetFont 的 Click 事件处理函数，首先创建一个 FontDialg 对象 fdlg，通过 Font 属性设置它的默认值为 btnSetFont 的字体，然后通过 ShowDialog()显示对话框。如果用户选择了新的字体，则将新字体设置为 btnSetFont 的字体。

示例代码 5-14

```
private void btnSetFont_Click(object sender, EventArgs e)
{
    FontDialog fdlg = new FontDialog( ); //创建 FontDialog 对象
    fdlg.Font = btnSetFont.Font;         //设置默认字体为 btnSetFont 当前字体
    if (fdlg.ShowDialog( ) == DialogResult.OK) //显示对话框，并等待返回
    {
        this.btnSetFont.Font = fdlg.Font; //选择了新的字体，则更新 btnSetFont
                                           的字体
    }
}
```



图 5-23　字体选择对话框效果图

5.6　小　　结

Windows 窗体是最常用的一种用户界面开发技术，窗体应用程序也是最常用的应用程序之一。窗体和控件是窗体应用程序最基本的两个元素，控件是最小的可视化元素，窗体是一个容器，它可以包含多个控件，通过对它们进行布局可以绘制出需要的用户界面。事

件机制则是窗体应用程序与用户进行交互的基础，通过事件可以随时监测到用户的操作，以及应用程序内部的变化。

本章从窗体应用程序、窗体、控件、对话框等 4 个方面介绍如何在 Visual Studio 2010 中进行窗体应用程序的开发，通过本章的学习，读者应该掌握以下知识点：

- 什么是窗体应用程序？
- 什么窗体和控件？
- 窗体的生命周期。
- 如何设置窗体的属性？如何处理窗体事件？
- 控件的公共特性有哪些？
- 如何使用 Label 控件进行文本显示？
- 如何使用 Button 按钮控件触发用户操作？
- 如何使用 TextBox、MaskedTextBox 控件获取用户输入的数据？
- 如何使用 CheckBox 和 RadioBuuton 进行候选项的选择？它们有什么区别？
- 如何使用 ListBox 和 ComboBox 实现列表中元素的选择？如何操作列表中的元素？
- 如何使用 MenuStrip 和 ContentMenuStrip 实现主菜单和上下文菜单？
- 如何使用 ToolStrip 实现工具栏？
- 如何使用 StatusStrip 实现状态栏？
- .NET 类库提供了哪些通用对话框？如何使用这些对话框？

第 6 章　多文档 Windows 窗体程序

多文档窗体是对窗体的一种扩展，它使得窗体不仅仅能够包含控件，也能包含一个或多个自定义的窗体，这些窗体可以同时存在和工作，可以相互切换，可以彼此间进行数据通信。多文档窗体应用程序为应用软件提供了一种更加灵活的用户界面表达方式，使得软件的开发和设计都更加灵活，本章将介绍在多文档窗体程序中会遇到的主要技术问题，以及如何解决它们。

6.1　创建多文档窗体程序

多文档窗体应用程序的主窗体是一个多文档窗体，它通常不包含任何控件，只用于包含一个或多个子窗体，这些子窗体实现具体的功能，它们彼此之间根据需要进行数据交互或者完全独立。本节将介绍多文档窗体程序的基础知识。

6.1.1　什么是多文档窗体程序

在诸如文本编辑器、图像处理器这样的应用软件中，通常需要同时处理一个或多个文档，每个文档独立地执行软件所需要的功能。这种需要在一个窗体中同时包含多个子窗体的应用程序通常成为多文档（MDI）应用程序，子窗体之间可以进行数据交互，也可以互不相干。如图 6-1 所示，Visual Studio 2010 开发环境就是多文档应用程序的典型实例。通常情况下，多文档窗体应用程序具有以下几个特点。

- 分级管理：所有窗体分成父-子两级。父窗体作为容器管理子窗体，一个父窗体可以有零个或多个子窗体。
- 独立显示：各子窗体根据需要独立地处理任何类型数据，并显示到界面，与用户进行交互。
- 并发处理：多个子窗体可以同时在后台处理数据，且相互之间可以进行交互。
- 容易修改：当某个功能需求发生变化时，只需要修改对应子窗体，方便快捷。
- 容易扩展：当需要增加某个功能时，只需要增加对应功能的子窗体，并添加到父窗体。

多文档应用程序的主窗体是一个能够包含其他窗体的窗体，通常称为父窗体，它是一个窗体容器，负责统一管理它所包含的子窗体。通常，各子窗体之间存在一些共同点（比如相似的界面风格、继承自同一个基类等），需要通过父窗体统一处理，子窗体之间也可能有一些共享数据、协作操作等。

在多文档应用程序中，父窗体可以管理子窗体，包括创建、删除、显示和隐藏子窗体。

当关闭父窗体时，自动关闭所有子窗体，然后退出应用程序。如果某一个子窗体关闭失败，则取消父窗体的关闭，已经被关闭的子窗体则不能恢复。

通常，父窗体包含所有子窗体都需要的菜单项，通过这些菜单项来管理子窗体，或引发子窗体的某些操作，如图 6-1 中开始、编辑、视图等菜单。子窗体也会根据特定需要制作专用菜单，然后合并到父窗体菜单中，变成一个统一的菜单项，如图 5-1 中的生成和调试菜单，就是在打开项目之后才集成到主菜单的。一般来说，父窗体包括窗口菜单项，在各子窗体之间进行切换，按不同方式排列子窗体。

子窗体是多文档程序的必要元素，它是用户交互和数据处理的核心，从理论上可以对不同类型数据（如数据库、用户输入等）进行各种操作（比如保存到文件、显示到界面、从数据库删除等）。多个子窗体中只能有一个是活动窗体，它获得用户输入焦点，与用户交互，进行前台数据处理。如图 6-1 所示，在 Visual Studio 2010 中，代码编辑窗口和窗体设计器是两个不同功能的子窗体，它们之间又相互关联，对代码的部分修改会映射到窗体设计器上。

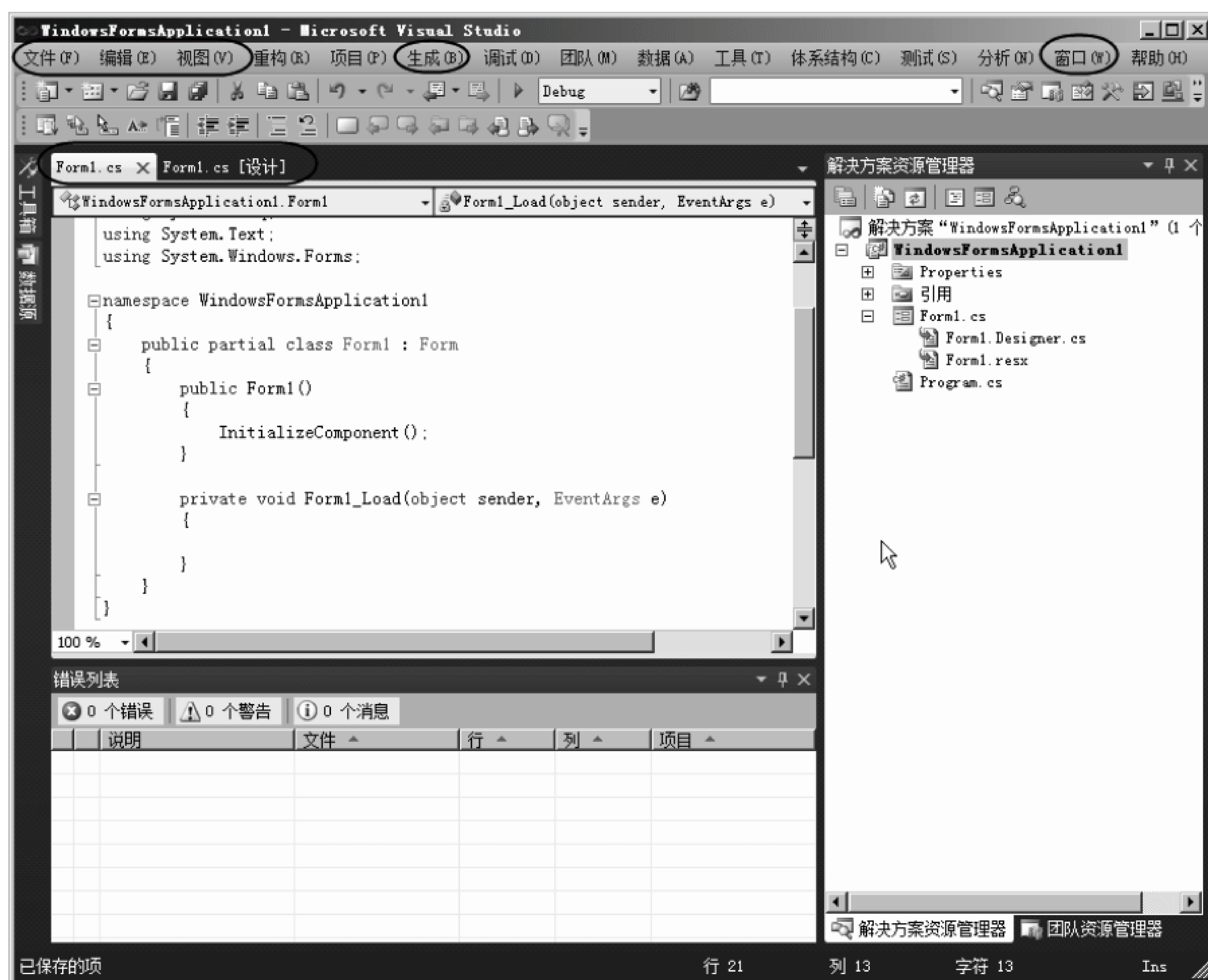


图 6-1　Visual Studio 多文档界面

提示： 多文档窗体设计器也不一定就必须有菜单，本节介绍的菜单项都是指多文档应用程序的基本结构，不同的应用程序对菜单的处理可能会不一样。

6.1.2　创建 Visual Studio 2010 提供的多文档父窗体

前面曾经讲到，在.NET 类库中，所有窗体都是由 System.Windows.Forms.Form 类实现，所以多文档窗体程序中的子窗体和父窗体都由 Form 类实现。当 Form 类的 IsMdiContainer

属性为 true 时表示该窗体为多文档窗体，它的用户区域默认为空白。多文档窗体的 MainMenu 属性所指向的菜单为应用程序主菜单，通过该菜单的 MdiWindowListItem 属性指明各子窗体标题被动态添加到哪个菜单项。

此外，Visual Studio 2010 为多文档窗体提供一个专门模板，通过它可以自动生成一个父窗体，该父窗体包含文件菜单、窗口菜单、工具菜单、工具栏、状态栏等基本控件。窗体设计人员可以在该窗体基础上进行修改得到适合自己需要的窗体。

假设现在已经新建了一个 Windows 窗体项目 DefMDIApp，在 Visual Studio 2010 中通过以下 3 个步骤创建一个多文档父窗体。

（1）右击“解决方案资源管理器”中的项目 DefMDIApps，在弹出菜单中选择“添加”|“Windows 窗体”命令，弹出“添加新项”对话框。

（2）在“模板”列表中选择“MDI 父级”，在“名称”文本框中输入窗体名称，cs 文件名后缀是自动添加的。这里输入 FrmMDIMain。

（3）单击“添加”按钮，看到新添加的窗体 FrmMDIMain，如图 6-2 所示。

从图 6-2 中可以看出，Visual Studio 2010 自动生成的父窗体包含了在文档编辑中常用的控件，包括文件菜单、编辑菜单、视图菜单、工具菜单、窗口菜单、帮助菜单，还带有工具栏和状态栏。在实际的开发中，可以根据需要修改这些基本控件，比如添加新菜单、修改现有菜单、删除现有菜单、增加工具栏按钮等，得到适合实际开发需要的多文档父窗体。其中最值得保留的是窗口菜单和视图菜单。本章后面章节会详细介绍父窗体的使用，此处不再赘述。

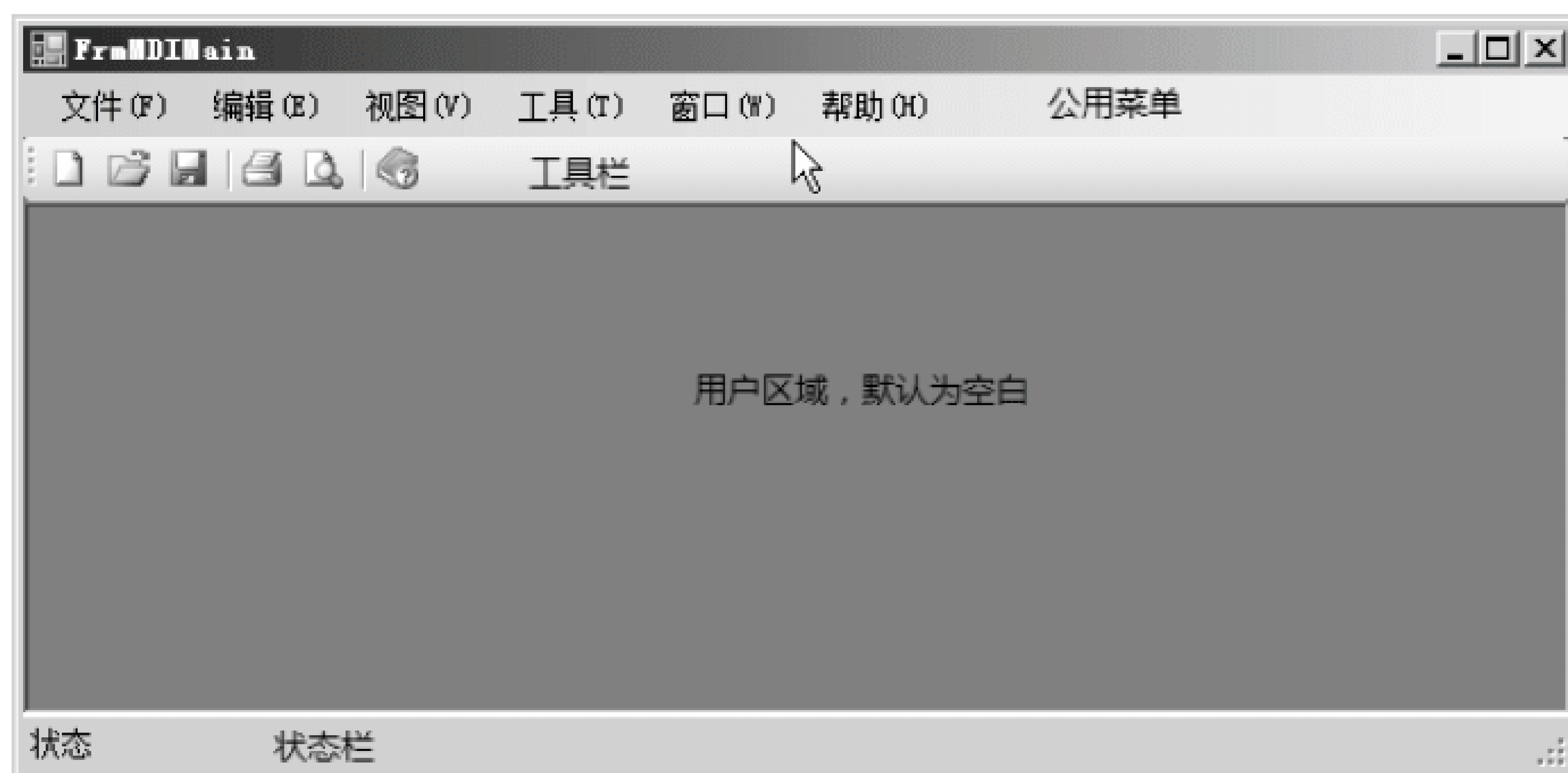


图 6-2　自动生成的 MDI 父窗体

注意：为了让新添加的 MDI 父窗体作为应用程序主窗体，需要将 Program.Main()方法中的代码“Application.Run(newForm1());”改成“Application.Run(new FrmMDIMain());”，同时可以删除没有用的窗体 Form1。

6.1.3　详细分析多文档父窗体的实现

在 6.1.2 节中创建的 FrmMDIMain 窗体是典型的多文档父窗体，它提供了多个典型的菜单和多文档父窗体界面布局。FrmMDIMain 的主菜单 Name 为 menuStrip 的菜单，它的 MdiWindowListItem 属性设置为它的一个名为“窗口”的菜单项，该菜单项提供进行窗体管理的常见命令，如图 6-3 所示。这些命令包括：新建窗口、层叠、垂直平铺、水平平铺、

全部关闭和排列图标，在 6.1.4 节将详细介绍它们的具体实现。

FrmMDIMain 的另外一个典型菜单是文件菜单，如图 6-4 所示，其中真正通用的是新建和退出菜单，新建菜单的功能与窗口菜单下的新建窗口通常是一样的。而退出菜单通常是调用 this.Close()关闭当前父窗体，从而关闭所有的子窗体。

另外，一个多文档父窗体的状态栏和工具栏通常是用户可选的，即用户可以选择是否需要这个功能。所以 FrmMDIMain 窗体提供视图菜单，通过它可以动态显示工具栏和状态栏，如图 6-5 所示。示例代码 6-1 为这两个菜单项的具体实现。

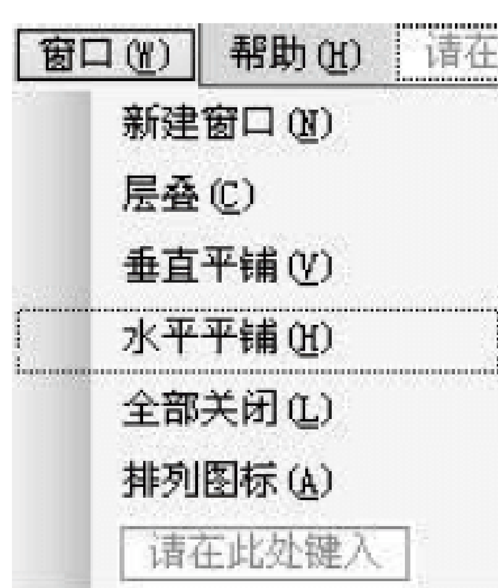


图 6-3　窗口菜单的常见命令



图 6-4　文件菜单的常见命令

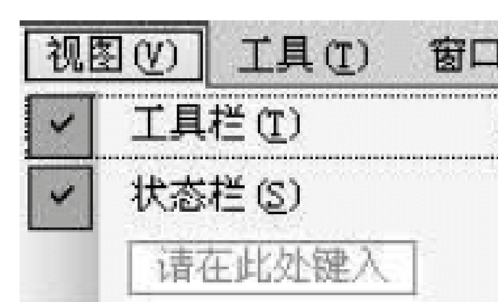


图 6-5　视图菜单命令

示例代码 6-1

```
private void ToolBarToolStripMenuItem_Click(object sender, EventArgs e)
{
    toolStrip.Visible = toolBarToolStripMenuItem.Checked;
                                                  //根据菜单项选中状态显示工具栏
}
private void StatusBarToolStripMenuItem Click(object sender, EventArgs e)
{
    statusStrip.Visible = statusBarToolStripMenuItem.Checked;
                                                  //根据菜单项选中状态显示状态栏
}
```

提示： 并非多文档父窗体就一定要有功能类似于窗口的菜单，也不一定要有文件菜单，这些只是.NET 类库提供的常见模式，真正的应用程序中会根据需要进行修改。

6.2　在父窗体中管理子窗体

管理子窗体是多文档父窗体的主要功能，通常包括添加、关闭和排列 3 个功能。本节将继续以 FrmMDIMain 为例，介绍这几个功能的具体实现。

6.2.1　添加子窗体到父窗体

在第 5 章介绍过，Form 类提供一个属性 MdiParent，用来获取或设置当前窗体的多文档父窗体。要为一个多文档父窗体添加子窗体主要有 3 个步骤：

（1）获取要添加的子窗体 childForm，新创建或从其他地方获取已经存在的窗体。

（2）将子窗体 childForm 的 MdiParent 属性设为当前多文档父窗体。

（3）显示子窗体 childForm。

示例代码 6-2 为 FrmMDIMain 窗体的菜单“文件”|“新建”菜单的具体实现，它首先创建一个新的子窗体 childForm，然后设置 childForm 的 MdiParent 属性为 this（即当前窗体），然后通过 Form.Show()方法显示子窗体。

示例代码 6-2

```
private void ShowNewForm(object sender, EventArgs e)
{
    Form childForm = new Form( );         //创建要添加的子窗体
    childForm.MdiParent = this;           //设置子窗体的 MdiParent 为当前窗体
    childForm.Text = "窗口 " + childFormNumber++;    //设置子窗体的标题
    childForm.Show( );                    //显示子窗体
}
```

当成功添加窗体到多文档父窗体之后，它的 MdiWindowListItem 所指向的菜单项下面会新增一个菜单项表示新增的窗体。所以执行示例代码 6-2 之后，会添加新菜单到窗口菜单下，如图 6-6 所示。

注意：“窗口”菜单下动态添加的表示子窗体的菜单项数量最多为 9 个，更多的情况下，可以通过“其他窗口”在一个列表中进行选择。

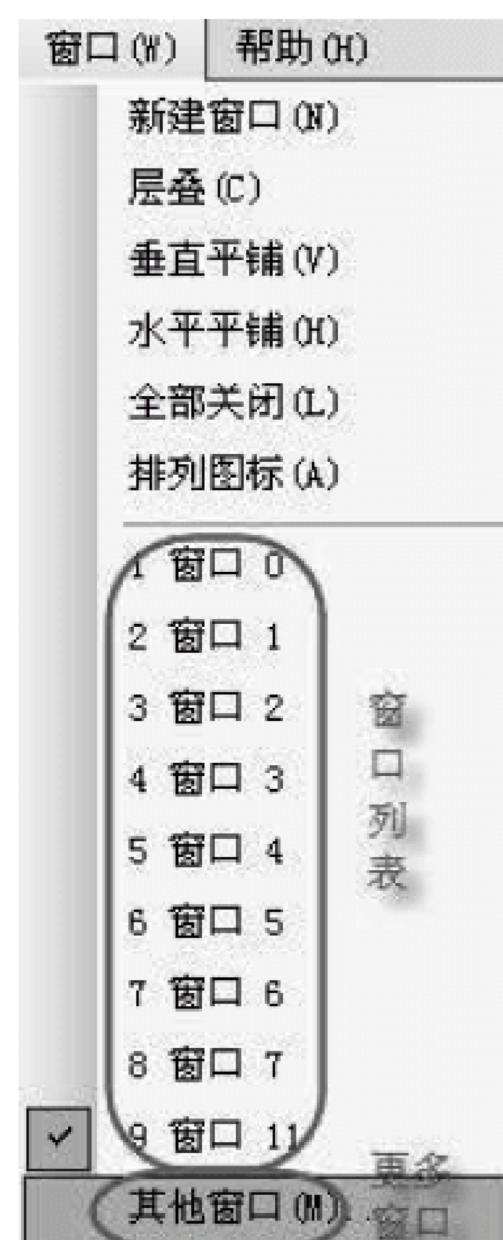


图 6-6 “窗口”新增窗体菜单

6.2.2　关闭打开的子窗体

要关闭某个子窗体，只需要在选中它的情况下，通过单击界面上右上角的“关闭”按钮来完成。也可以通过 Form 的 ActiveMdiChild 来获取当前活动的子窗体 childForm，然后通过调用 childForm 的 Close()方法关闭它。

如示例代码 6-3 为 FrmMDIMain 中“关闭当前窗体”菜单项的执行代码，首先通过 ActiveMdiChild 属性获取父窗体（this）中的活动窗体 childForm，如果活动窗体存在，则通过 Close()方法关闭它。

示例代码 6-3

```
private void tsmiCloseCurFrm Click(object sender, EventArgs e)
{
    Form childForm = this.ActiveMdiChild;       //获取当前活动的子窗体
    if (childForm != null)
    {
        childForm.Close( );                     //关闭子窗体
    }
}
```

🔔**技巧：** 如果需要关闭全部子窗体并退出应用程序，则可以通过 Form.Close()方法关闭多文档父窗体，间接关闭它的所有子窗体，然后退出应用程序。

6.2.3　遍历存在的子窗体

Form 类提供属性 MdiChildren，它是一个 Form 类型数组，用来获取当前父窗体所包含的所有子窗体，通过遍历该集合可以找到当前父窗体中的所有子窗体。

示例代码 6-4 为 FrmMainMDI 的“窗口”|“全部关闭”命令的具体实现，首先通过 MdiChildren 属性获取当前父窗体中的所有子窗体，然后通过 Close()方法关闭这些子窗体。

示例代码 6-4

```
private void CloseAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    foreach (Form childForm in MdiChildren)     //遍历所有的子窗体
    {
        childForm.Close( );                     //关闭子窗体
    }
}
```

🔔**注意：** 窗体遍历不仅仅用于关闭窗体，当父窗体需要将数据传递到子窗体时，也通常需要遍历子窗体，然后进行相应的处理。在 6.3 节中将介绍更多细节。

6.2.4　排列存在的子窗体

在使用多文档窗体程序的时候，通常会出现多个子窗体，有时为了同时浏览多个子窗体的数据，需要对这些窗体进行排列。在.NET 类库中，Form 类提供 LayoutMdi()方法，用来排列多文档父窗体中的多个子窗体，LayoutMdi()方法的声明如下：

```
void LayoutMdi(MdiLayout value)
```

其中，参数 value 是 MdiLayout 枚举类型，用来表明如何排列多个子窗体，有以下 4 个可选项。

- Cascade：将多个窗体层叠排布在 MDI 父窗体工作区内。
- TitleHorizontal：按标题水平平铺排列在 MDI 父窗体工作区内。
- TitleVertical：按标题垂直平铺排列在 MDI 父窗体工作区内。
- ArrangeIcons：按 MDI 子图标均排列在 MDI 父窗体工作区内。

在窗体 FrmMDIMain 中，通过“窗口”菜单提供了 4 种排列方式，分别是层叠、水平平铺、垂直平铺、排列图标。这 4 个功能的具体实现如示例代码 6-5 所示，它们分别为方法 LayoutMdi()传递对应的参数来达到对多个子窗体进行排序的功能。图 6-7、图 6-8、图 6-9 分别是 FrmMDIMain 包含 3 个子窗体时按照层叠、水平平铺、垂直平铺排列的效果图。

示例代码 6-5

```
private void CascadeToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    LayoutMdi(MdiLayout.Cascade);                  //层叠排列
}
private  void  TileVerticalToolStripMenuItem_Click(object  sender,  Event
Args e)
{
    LayoutMdi(MdiLayout.TileVertical);             //垂直平铺排列
}
private void TileHorizontalToolStripMenuItem Click(object sender, Event
Args e)
{
    LayoutMdi(MdiLayout.TileHorizontal);           //水平平铺排列
}
private  void  ArrangeIconsToolStripMenuItem_Click(object  sender,  Event
Args e)
{
    LayoutMdi(MdiLayout.ArrangeIcons);             //排列图标
}
```



图 6-7　层叠排列的窗体

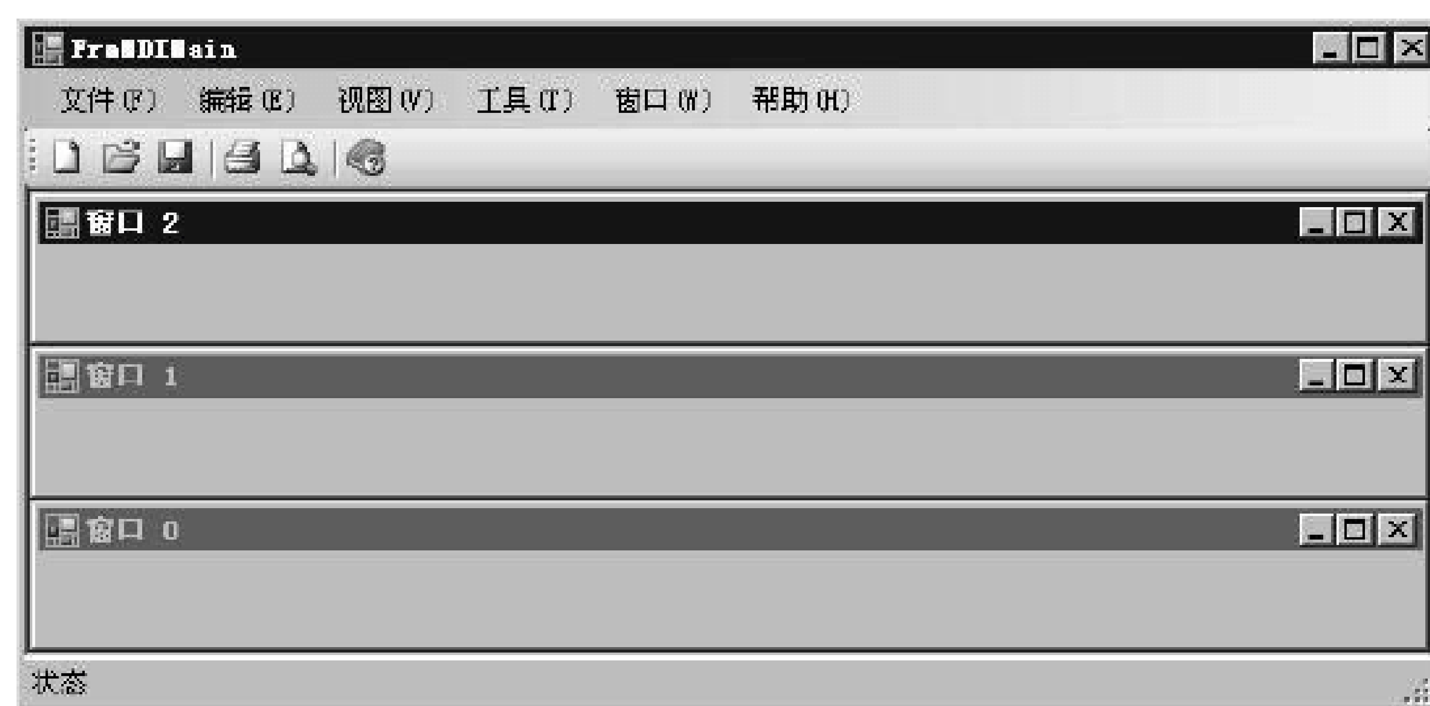


图 6-8　水平平铺排列的窗体



图 6-9　垂直平铺排列的窗体

6.3　文件阅读器实例

本节通过一个基于多文档窗体的文本文件阅读器实例，较全面地介绍多文档应用程序开发中比较重要的问题，包括窗体间数据交互、管理窗体等。

6.3.1　创建文本编辑器实例

在进一步介绍多文档应用程序开发细节之前，首先需要创建文件阅读器实例——

MultiTextReader 的主要框架。通过以下步骤建立该应用程序。

（1）打开 Visual Studio 2010，创建名为 MultiTextReader 的 Windows 窗体应用程序。

（2）删除默认生成的主窗体 Form1。

（3）通过添加向导，添加一个名为 FrmMain 的多文档父窗体，并修改 Program.Main()方法，使得 FrmMain 作为主窗体显示。

（4）移除 FrmMain 中部分不需要的菜单项，最后只留下表 6-1 所示的菜单项。

表 6-1　文本阅读器主菜单

菜　　单	菜 单 文 本	功 能 说 明
openToolStripMenuItem	打开	用于打开要查看的文本文件
exitToolStripMenuItem	退出	退出应用程序
toolBarToolStripMenuItem	工具栏	设置显示或隐藏工具栏
statusBarToolStripMenuItem	状态栏	设置显示或隐藏状态栏
optionsToolStripMenuItem	选项	设置显示文本数据所采用的选项
windowsMenu	窗口	各种排列窗口命令和已打开的窗口列表
aboutToolStripMenuItem	关于	显示关于对话框

（5）移除 FrmMain 工具栏中不需要的工具栏项，最后只保留打开功能。

（6）添加窗体 FrmReader 用于显示文件中的文本信息，它只包括一个只读的文本框 tbTexts。其中 tbTexts 的属性设置如表 6-2 所示。

表 6-2　FrmReader中tbTexts的属性设置

属　性　名	属　性　值	说　　明
Dock	Fill	表示该文本框占用窗体的整个用户区域
Multiline	true	表示该文本框可以显示多行数据
ReadOnly	true	表示该文本框为只读，不可编辑它的内容
ScrollBars	Both	表示该文本框同时具有水平和垂直滚动条

（7）通过向导添加一个模板为“关于框”的对话框 AboutBox，它自动从 AssemblyInfo.cs 获取该文本阅读器的公司、版本等信息，并显示到界面。

（8）为菜单和工具栏添加事件处理函数，这里主要介绍“关于”菜单的 Clcik 事件处理函数。如示例代码 6-6 所示，首先创建一个 AboutBox 对象，然后通过 ShowDialog()方法显示“关于”对话框，产生如图 6-10 所示的对话框。

示例代码 6-6

```
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
    AboutBox dlg = new AboutBox( );
    dlg.ShowDialog(this);
}
```

提示：“关于框”模板提供的“关于”对话框自动从 AssemblyInfo.cs 获取软件信息，所以只需要更新这个代码文件即可，不需要对 AboutBox 进行代码上的修改，可以大大节省开发时间。

6.3.2　打开文件阅读子窗体



图 6-10 “关于”对话框效果图

在打开文件进行阅读之前，首先需要保证子窗体具有查看文件文本内容的功能，子窗体 FrmReader 负责打开和显示文本文件中的内容。如示例代码 6-7 所示，FrmReader 通过构造函数获取要显示文件的文件名，并用 string 类型字段_FileName 保存它。在窗体第一次显示时，在 Load 事件处理函数 FrmReader_Load()中设置窗体的标题为正在查看的文件名，通过 System.IO.StreamReader 类依次读取文件中的文本，并显示到文本框 tbTexts 中。

示例代码 6-7

```
public partial class FrmReader : Form
{
    public FrmReader(string fileName)
    {
        this._FileName = fileName;                //更新当前打开文件的文件名
        InitializeComponent( );
    }
    //表示当前子窗体所打开文件的文件名
    private string  _FileName;
    //窗体加载时显示加载并显示文件内容
    private void FrmReader_Load(object sender, EventArgs e)
    {
        this.Text = this._FileName;               //设置当前窗体的标题
        StreamReader sr;                          //文件流对象 sr
        sr = new StreamReader(this._FileName, Encoding.Default);
                                                  //打开指定文件
        while  (!sr.EndOfStream)                  //如果文件没有读取完成继续读取
        {
            string line = sr.ReadLine( );         //从文件读取一行
            this.tbTexts.AppendText(line);        //显示文件数据到文本框
            this.tbTexts.AppendText(System.Environment.NewLine);
        }
        sr.Close( );
    }
}
```

同时，在父窗体 FrmMain 中，打开一个文件时需要做两件事，首先通过 OpenFileDialog 对话框获取要打开的文件路径，然后创建并显示一个 FrmReader 子窗体。如示例代码 6-8 所示，通过 OpenFileDialog 类的 Filter 属性指定只能打开文本文件（扩展名为.txt）。另外，在创建 FrmReader 对象 frm 时，将文件路径 fileName 作为参数传递到子窗体 frm 中。

示例代码 6-8

```
private void OpenFile(object sender, EventArgs e)
{
```

```
    OpenFileDialog ofdlg = new OpenFileDialog( );  //创建OpenFileDialog对
                                                     象 ofdlg
    //设置打开文件对话框的默认路径为“我的文档”
    ofdlg.InitialDirectory = Environment.GetFolderPath(Environment.
    SpecialFolder.Personal);
    ofdlg.Filter = "文本文件(*.txt)|*.txt";       //设置只接受txt为扩展名的文件
    if (ofdlg.ShowDialog(this) == DialogResult.OK) //显示打开文件对话框
    {
        string fileName = ofdlg.FileName;         //获取要打开的文件名
        FrmReader frm = new FrmReader(fileName);//创建查看文件的子窗体 frm
        frm.MdiParent = this;                   //设置子窗体 frm 的父窗体为当前窗体
        frm.Show( );                              //显示子窗体
    }
}
```

技巧： 通过窗体的方法或构造函数进行数据交互是窗体间数据交互的一种常用方式。方法的参数通常是窗体接受外部的数据，而方法的返回值则通常是窗体为外部提供数据。

图 6-11 为文本阅读器打开两个文件“C:\新建文本文档.txt”和“C:\祝福短信.txt”之后的运行效果图，从“窗口”菜单中可以看出正在阅读的文件列表，从窗体标题栏可以看出应用软件的标题（前部分）和当前打开窗体的标题（后部分）。

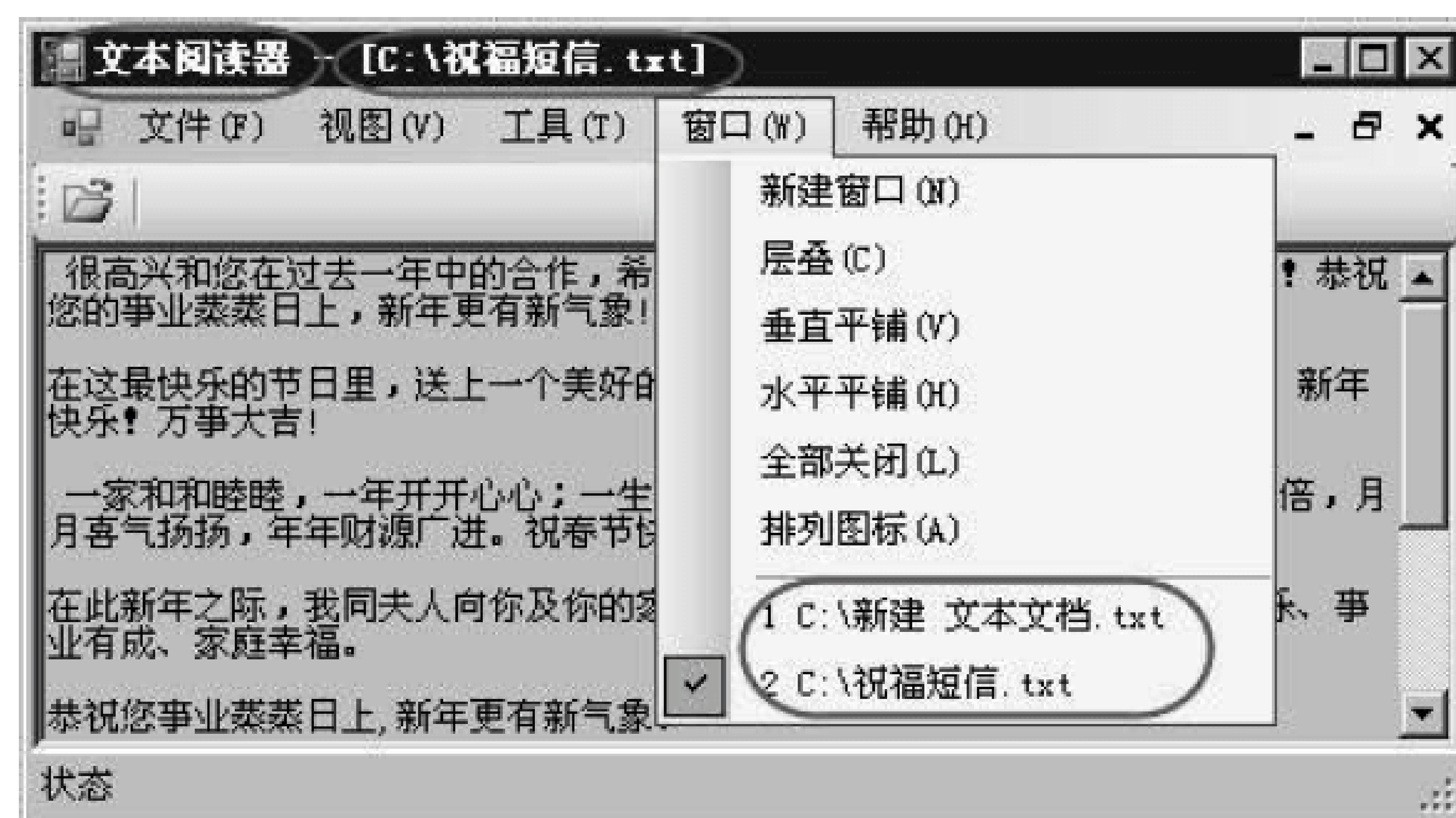


图 6-11　文本阅读器效果图

6.3.3　设置阅读参数对话框实现

每个用户都有自己阅读的偏好，比如张三喜欢红色，而李四喜欢黑色，这些参数通常是通过对话框的形式让用户进行设置，并更新到已经打开的子窗体中。在实例 MultiTextReader 中，要创建一个新的窗体 SettingDlg，用来设置 3 个基本的现实文本的参数：前景色、背景色和字体。

SettingDlg 包括 3 个基本字段_BackColor、_ForeColor 和_Font，分别保存查看文件内容时使用的背景色、前景色和字体。另外，在构造函数中通过 3 个参数接受默认的查看参数，并且在窗体加载时通过 Load 事件处理函数刷新界面。在前景色、背景色和字体的设置按钮 Click 事件处理函数中，分别使用 ColorDialog 和 FontDialog 获取新的颜色和字体，并更新对应的字段和界面，如示例代码 6-9 所示。

示例代码 6-9

```
public partial class SettingDlg : Form
{
    //构造函数，接受 3 个参数，查看背景色、前景色、字体
    public SettingDlg(Color bkcolor, Color forecolor, Font fnt)
    {
        this._BackColor = bkcolor;                //更新背景色字段
        this._ForeColor = forecolor;              //更新前景色字段
```

```
    this. Font = fnt;                           //更新字体字段
    InitializeComponent( );
}
private Color _BackColor;                       //背景色字段
public Color GetBackColor( )                    //获取新设置的背景色
{
    return this. BackColor;
}
private Color _ForeColor;                       //前景色字段
public Color GetForeColor( )                    //获取新设置的前景色
{
    return this. ForeColor;
}
private Font _Font;                             //字体字段
public Font GetFont( )                          //获取新设置的字体
{
    return this. Font;
}
private void RefreshSample( )                   //根据最新的参数刷新界面
{
    this.lbBackColor.BackColor = this._BackColor;   //背景色
    this.lbForcolor.BackColor = this. ForeColor;    //前景色
    this.lbFont.Font = this._Font;                  //字体
    this.tbSample.BackColor = this._BackColor;      //更新示例文本参数
    this.tbSample.ForeColor = this. ForeColor;
    this.tbSample.Font = this. Font;
}
//窗体加载时，将默认的查看参数刷新到界面
private void SettingDlg_Load(object sender, EventArgs e)
{
    RefreshSample( );
}
//单击修改前景色按钮，选择新的前景色，并更新界面
private void btnForeColor Click(object sender, EventArgs e)
{
    ColorDialog dlg = new ColorDialog( );       //选择颜色对话框
    dlg.Color = this._ForeColor;                //设置默认颜色
    if (dlg.ShowDialog( ) == DialogResult.OK)
    {
        this._ForeColor = dlg.Color;            //更新参数和界面
        RefreshSample( );
    }
}
//单击修改背景色按钮，选择新的背景色，并更新界面
private void btnBackColor Click(object sender, EventArgs e)
{
    ColorDialog dlg = new ColorDialog( );       //选择颜色对话框
    dlg.Color = this._BackColor;                //设置默认颜色
    if (dlg.ShowDialog( ) == DialogResult.OK)
    {
        this. BackColor = dlg.Color;            //更新参数和界面
        RefreshSample( );
    }
}
//单击修改字体按钮，选择新的字体，并更新界面
private void btnFont_Click(object sender, EventArgs e)
{
    FontDialog dlg = new FontDialog( );         //选择字体对话框
```

```
        dlg.Font = this. Font;                              //设置默认字体
        if (dlg.ShowDialog( ) == DialogResult.OK)
        {
            this._Font = dlg.Font;                          //更新参数和界面
            RefreshSample( );
        }
    }
    //单击“确认”按钮，返回 OK
    private void btnOk Click(object sender, EventArgs e)
    {
        this.DialogResult = DialogResult.OK;
    }
    //单击“取消”按钮，返回 Cancel
    private void btnCancel Click(object sender, EventArgs e)
    {
        this.DialogResult = DialogResult.Cancel;
    }
}
```

如图 6-12 为“阅读参数设置”对话框的效果图。



图 6-12　阅读参数设置效果图

🔔注意：在开发这种参数设置功能的对话框时，在第一次加载对话框时能够显示当前正在使用的参数，并且可以取消设置新参数，而且最好能给出新参数的示例。

6.3.4　更新阅读参数到子窗体

在实现了对话框 SettingDlg 之后，为了可以使用它进行阅读参数的设置，在多文档窗体 FrmMain 中需要创建和使用该对话框，然后将新的阅读参数更新到已有的子窗体中。

如示例代码 6-10 所示，FrmMain 新增 3 个字段_ViewBackColor、_ViewForeColor 和_ViewFont，分别保存阅读时采用的背景色、前景色和字体，并在定义时给出默认值，防止产生异常。在“选项”菜单处理函数 optionsToolStripMenuItem_Click()中，首先，用当前的阅读参数作为 SettingDlg 的构造函数参数创建一个 SettingDlg 对话框 dlg。然后，显示对话框并等待用户设置新参数，最后，通过 MdiChildren 属性遍历当前已经存在的所有子窗体，如果子窗体为 FrmReader 则更新它的阅读参数。

示例代码 6-10

```
public partial class FrmMain : Form
{
    private Color  ViewBackColor = Color.White;//阅读的背景色字段及默认值
    private Color _ViewForeColor = Color.Black;//阅读的前景色字段及默认值
    private Font _ViewFont = SystemFonts.DefaultFont;//阅读的字体字段及默认值
    //“选项”菜单处理函数，打开 SettingDlg，并将新参数更新到已打开的子窗体
    private void optionsToolStripMenuItem_Click(object sender, EventArgs e)
    {
        SettingDlg dlg = new SettingDlg(this._ViewBackColor,
                                    //用当前阅读参数创建 SettingDlg 对象 dlg
                                  this. ViewForeColor,
                                  this._ViewFont);
```

```
        if (dlg.ShowDialog(this) == DialogResult.OK)
                                                //显示参数设置对话框，并等待用户设置
        {
            this._ViewBackColor = dlg.GetBackColor( );
                                                //获取用户最新的阅读参数
            this._ViewFont = dlg.GetFont( );
            this._ViewForeColor = dlg.GetForeColor( );
            foreach (Form frm in this.MdiChildren)   //遍历所有子窗体
            {
                FrmReader frmRd = frm as FrmReader;   //如果子窗体是 FrmReader
                                                        类型
                if (frmRd != null)                      //则更新最新参数到子窗体
                {
                    frmRd.SetViewSettings(this._ViewBackColor,
this._ViewForeColor, this._ViewFont);
                }
            }
        }
    }
}
```

6.4　小　　结

Windows 多文档应用程序是一种常见的应用程序类型，它提供一个父窗体，同时包含一个或多个子窗体，每个子窗体实现各自的功能，从而实现功能复杂多样的软件。在.NET 类库中，通过窗体类（Form）同时提供父窗体和子窗体的支持，开发人员根据需要编写特定类型的窗体。

本章通过一个典型的文件阅读器实例给读者介绍了多文档窗体的相关知识，通过本章的学习读者应该掌握以下知识点：

- 什么是多文档窗体和多文档应用程序？
- 如何通过 Form 类实现多文档窗体？
- 父窗体如何管理多个子窗体？
- 窗体之间如何进行数据交互？

第 7 章　.NET 类库开发

在实际开发的应用软件开发中，通常需要将一个应用软件按照不同的设计思路分解成多个模块，每个模块是一个独立的动态链接库。这样既可以达到尽可能多的重用，也可以让应用软件的结构和思路更加清楚。在 Visual Studio 2010 中除了可以开发出本书前面示例中的独立应用程序之外，还可以通过创建.NET 类库应用程序生成独立的动态链接库，本节将重点介绍.NET 类库的开发。

7.1　.NET 类库项目

.NET 类库项目是在.NET 框架下开发动态链接库（DLL）的项目，它也是模块划分和.NET 类库的基础，本节将介绍如何通过 C#开发.NET 类库项目。

7.1.1　什么是.NET 类库

在.NET 框架下，.NET 类库既可以包含类、接口、结构体、枚举等自定义数据类型，也可以包含 Windows 窗体、Windows 自定义控件、字符串资源等资源。一个.NET 类库通常包含在同一个项目（Project）中，该项目包含类库所有的实现代码。注意，本章的.NET 类库是指.NET 开发的动态链接库，不要和微软提供的.NET 类库混为一谈。

通常，一个.NET 类库对外提供一个或多个自定义数据类型，这些数据类型有的需要公开出来让应用程序使用，有的只是在 DLL 内部使用。通过访问修饰关键字 public、private、internal、protected 控制这些数据类型的对外可见性。其中通常用到以下两个。

- public：表示该数据类型对外部应用程序公开，应用程序可以不受限制使用这些数据类型。
- internal：表示该数据类型不对外公开，只在相同命名空间下可访问。

通常，一个.NET 类库具有一个根命名空间，用来访问该动态链接库下的所有数据类型和资源。同样一个动态链接库也可以在该根命名空间下根据需要再细分出很多命名空间分支。这些命名空间的命名通常具有实际意义，根据功能进行分支。在创建一个.NET 类库之前要先明确以下 4 个问题。

（1）为什么要创建.NET 类库？它有哪些功能？

通常，.NET 类库是一个相对独立和完整的功能模块的封装，它对外提供必要的功能接口，并隐藏具体的内部实现细节。所以在开发一个 DLL 之前，必需明确它的功能。

（2）.NET 类库如何命名？它包含多少命名空间？

在.NET 环境下，.NET 类库是多个命名空间集合，它通常包含一个根命名空间和多个

子命名空间。.NET 类库的文件名称通常是根据开发者的希望来命名，能表示 dll 的用途，如 mydll.dll 表示一个自定义 DLL。命名空间名称通常按照功能命名，将动态链接库的数据从逻辑上分成多个子模块，比如："MyDll.Data" 表示 "mydll.dll" 的数据部分，"MyDll.UI" 表示 "mydll.dll" 的界面部分。

（3）.NET 类库有哪些对外接口？

一般地，.NET 类库要求对外接口简洁、全面、稳定，所以在开发之前要明确这个动态链接库需要对外接口，并且在编码的时候尽可能保持不变。

（4）.NET 类库的内部功能如何实现？

通常，一个类库不会是从零开始的，先将具体功能和已有的类库（.NET 系统类库和已有自定义类库）进行综合考虑，或是直接继承原有类库，或是扩展原有类库，或是借鉴原有类库，这些都要有个明确的实现方案。

明确了上面 4 个问题，基本上对要创建的.NET 类库就有了全面而深入的理解，再开始编码工作会更加得心应手和目标明确。

7.1.2 创建.NET 类库 AnimalLib

Visual Studio 2010 提供了 3 套模板用来开发不同类型的.NET 类库项目，它们是.NET 类库、Windows 控件库和 Web 控件库，通过这 3 个模板可以大大提高开发.NET 动态链接库的效率。

- .NET 类库：普通.NET 动态链接库，通常提供普通自定义数据类型。
- Windows 控件库：常用于创建提供公用 Windows 控件、Windows 窗体等的动态链接库。
- Web 控件库：常用于创建提供公用 Web 控件的动态链接库。

本节将介绍一个.NET 实例 AnimalLib，该类库需要为外部调用者提供一个表示动物的基类——Animal 类，并提供一个实现 Cat 类。在 Visual Studio 2010 中要创建 AnimalLib 类库需要下面两步：

（1）打开 Visual Studio 2010 开发环境，通过"开始" | "新建" | "项目" 菜单命令打开"新建项目" 对话框。

（2）在 C#项目的"模板" 列表中选择"类库"，在名称文本框中输入 AnimalLib。单击"确定" 按钮创建 AnimalLib 类库项目。

Visual Studio 2010 为 AnimalLib 项目自动生成最基本的程序框架，从"解决方案资源管理器"中可以看出它的代码结构，与前面介绍的应用程序项目不同，它不包含 Program.cs 代码文件，因为它本身不能直接执行。但是 AnimalLib 项目包含一个默认生成的 Class1.cs 文件，该文件提供一个空的类 Class1，通常根据需要删除或修改 Class1 类。

7.1.3 实现.NET 类库 AnimalLib

在 7.1.1 节介绍了实现一个.NET 类之前必须要考虑的问题，所以在实现 AnimalLib 类库之前首先要确定它的功能和接口。AnimalLib 类库需要实现两个基本的类：

- Animal 类：该类是所有动物的基类，是一个抽象类，只提供基本的函数定义和公

用代码，但是它需要公开给调用者，所以应该为 Public 的。

- Cat 类：该类是一个 Animal 的子类，用来表示“猫”这种动物，它也是需要对外公开。

实现.NET 类库和实现普通的应用程序一样，根据需要编写对应的类和代码，只是需要注意命名空间和可访问性即可，示例代码 7-1 给出了 Animal 类的具体实现。

首先，需要删除自动生成的 Class1 类。然后，通过“解决方案资源管理器”添加一个名为 Animal 的类，并为它提供必要的成员：Name 属性、Running()方法和 Eating()方法。另外，在同一个代码文件 Animal.cs 中实现另外一个类 Cat，它从 Animal 类继承，而且重写 Running()方法和 Eating()方法，还增加了属性 Weight。

示例代码 7-1

```
namespace AnimalLib
{
    //Animal 类提供最基本的动物的实现，Name 属性，Running()方法和 Eating()方法
    public abstract class Animal
    {
        //string 类型，表示名称的字段和属性
        private string  Name;
        public string Name
        {
            get
            {
                return this. Name;
            }
            set
            {
                this. Name = value;
            }
        }
        //虚函数 Running()提供最基本的实现
        public virtual void Running()
        {
            System.Console.WriteLine("Animal {0} is running", this. Name);
        }
        //虚函数 Eating()提供最基本的实现
        public virtual void Eating()
        {
            System.Console.WriteLine("Animal {0} is eating", this._Name);
        }
    }
    //Cat 类表示“猫”，从 Animal 类继承而来
    public class Cat : Animal
    {
        //int 类型，表示体重的字段和属性，在设置体重时如果值不合理，则抛出异常，提示调
         用者
        private int _Weight;
        public int Weight
        {
            get
            {
                return this._Weight;
            }
            set
            {
```

```
                if (this. Weight <= 0)    //如果体重不合法，抛出异常
                    throw new SystemException("猫的体重不能小于等于零。");
                this._Weight = value;
            }
        }
        //重写 Animal.Eating()方法
        public override void Eating( )
        {
            System.Console.WriteLine("Cat {0} is eating", this.Name);
        }
        //重写 Animal.Running()方法
        public override void Running( )
        {
            System.Console.WriteLine("Cat {0} is running", this.Name);
        }
    }
}
```

注意：从示例代码 7-1 中可以看出，.NET 类库的根命名空间和.NET 类库项目的名称默认是一致的，除非通过项目属性修改了项目的根命名空间，这在项目比较大的时候很有必要。

7.1.4 使用.NET 类库 AnimalLib

通过前面两节的工作，基本上已经完成了.NET 类库 AnimalLib 的编码工作，在使用该类库之前还需要通过 AnimalLib 项目生成动态链接库文件——AnimalLib.dll。在 Visual Studio 2010 中只需要通过以下两个步骤即可完成这个操作：

（1）在“解决方案资源管理器”中右击项目 AnimalLib，在弹出的快捷菜单中选择“生成”或“重新生成”选项，生成项目。

（2）成功生成项目之后，可以在项目的输出目录下看到目标文件 AnimalLib.dll，通常在项目目录的 Debug 文件夹下。

注意：“生成”会检查上一次生成之后代码文件是否发生更改，而且只编译和已经修改过的代码，这在代码量巨大且编译时间过长时可以大大减少生成时间。“重新生成”则先删除所有生成的文件，重新生成所有代码，往往需要更长的时间。

在生成了 AnimalLib.dll 之后，通常有两种方法使用 AnimalLib.dll，它们也是使用其他.NET 类库的通用方法。

- 通过项目引用：在有动态链接库的项目源代码时，可以将应用程序项目和 DLL 项目添加到同一个解决方案中，通过项目来引用 DLL，这样可以随时使用最新版本的 DLL，而且可以对 DLL 的代码进行跟踪和调试。
- 通过 DLL 引用：在只有动态链接库文件，但没有源代码时，只能通过直接加载 DLL 文件为项目引用，常用的是使用第 3 方类库的方法。由于没有源代码，所以也不能对 DLL 内部的代码进行跟踪和调试。

下面介绍如何通过 DLL 引用.NET 类库 AnimalLib.dll，首先创建一个新的控制台应用程序，在 Visual Studio 2010 中，通过下面 3 个步骤来完成。

（1）打开 Visual Studio 2010，并新建一个名为 UseAnimalLib 的控制台应用程序。

（2）在“解决方案资源管理器”中右击项目 UseAnimalLib 下的“引用”子结点，在弹出菜单中选择“添加引用”命令，弹出“添加引用”对话框，如图 7-1 所示。选择“浏览”选项卡，然后找到 AnimalLib 项目的输出目录（通常为 Bin\Debug），选择 AnimalLib.dll 文件，然后单击“确定”按钮添加引用。

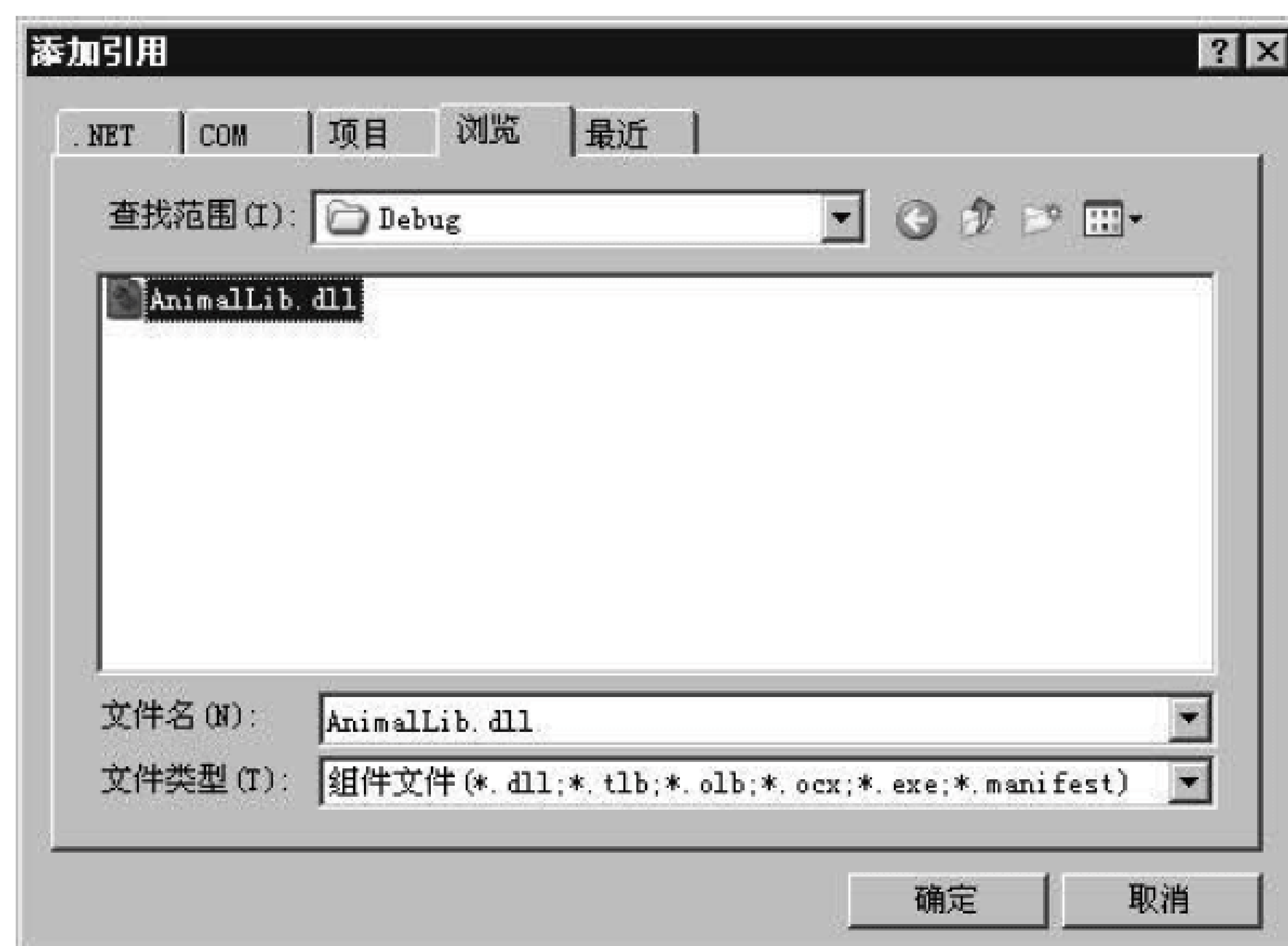


图 7-1　添加引用 AnimalLib.dll

（3）要在代码中能使用 AnimalLib.dll 类库里面的成员，还需要通过 using 语句来导入命名空间 AnimalLib，代码如下：

```
using AnimalLib;
```

通过以上 3 个步骤，完成 UseAnimalLib 项目对 AnimalLib.dll 类库的引用，UseAnimalLib 项目的“引用”子结点下会增加一个 AnimalLib 结点，表示项目中已经添加了“AnimalLib 类库”的引用。

示例代码 7-2 演示如何在项目 UseAnimalLib 中使用 AnimalLib.dll 提供的类型和接口，首先通过 using 子句引用命名空间 AnimalLib，然后像使用普通类一样使用 AnimalLib 下公开的类和接口。

示例代码 7-2

```
using AnimalLib;                              //引用 AnimalLib 命名空间
namespace UseAnimalLib
{
    //Cow 从 Animal 类继承而来，重写了 Eating()方法
    class Cow : Animal
    {
        public override void Eating( )
        {
            System.Console.WriteLine("Cow {0} is eating", this.Name);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            Cow aCow = new Cow( );          //创建一个 Cow 对象 aCow
            aCow.Name = "CowBB";            //设置 aCow 的 Name 属性
            aCow.Running( );                //依次调用 Running()和 Eating()方法
            aCow.Eating( );
            Cat aCat = new Cat( );          //创建一个 Cat 对象 aCat
            aCat.Name = "CatAA";            //设置 aCat 的 Name 属性
            aCat.Running( );                //依次调用 Running()和 Eating()方法
            aCat.Eating( );
        }
    }
}
```

示例代码 7-2 的输出如下，从中可以看出，由于 Cow 类并没有重写 Animal.Running()方法，所以 aCow.Running()实际是执行的 Animal.Running()方法。而 Cat 类重写了 Animal.Running()方法，所以 aCat.Running()执行的是 Cat.Running()方法。

```
Animal CowBB is running
Cow CowBB is eating
Cat CatAA is running
Cat CatAA is eating
```

7.1.5 通过项目引用 AnimalLib

在 7.1.4 节讲到，除了通过 DLL 引用的方式引用.NET 类库外，在有类库源代码的情况下，还可以将.NET 类库项目添加当前解决方案，然后通过项目引用该.NET 类库。本节将介绍这种方法的具体步骤，如要在应用程序 UseAnimalLib 中引用 AnimalLib.dll，还可以通过如下 3 步来完成。

（1）打开 UseAnimalLib 项目，并通过菜单添加现有项目 AnimalLib 到解决方案中。

（2）在“解决方案资源管理器”中右击项目 UseAnimalLib 下的“引用”子结点，在弹出的快捷菜单中选择“添加引用”选项，弹出“添加引用”对话框，如图 7-2 所示。在其中选择“项目”选项卡，这里会列出当前解决方案中可用的所有项目，选中 AnimalLib 项目，然后单击“确定”按钮添加引用。



图 7-2 添加引用项目 AnimalLib

（3）要在代码中能使用 AnimalLib.dll 类库里面的成员，还需要通过 using 语句导入命

名空间 AnimalLib。

这样，项目 UseAnimalLib 就依赖于项目 AnimalLib，所以在生成 UseAnimalLib 时，Visual Studio 2010 会自动检测项目 AnimalLib 是否发生更改，如果有更改，则自动生成项目 AnimalLib，这样就可以保证 UseAnimalLib 随时都可以使用最新的 AnimalLib.dll。

注意： 应用程序 UseAniamlLib.exe 运行时需要使用 AnimalLib.dll，所以生成之后 Visual Studio 2010 会自动将 AnimalLib.dll 复制到 UseAnimalLib 的输出目录。在发布应用程序时也要将 AnimalLib.dll 一起发布。

7.2　.NET 控件库

在微软官方提供的.NET 类库中，提供了大量用户控件，这些用户控件都是通过类库的方式提供的，在 Visual Studio 2010 中，.NET 控件库提供专门的用户自定义控件的集合，本节将详细介绍它的实现。

7.2.1　自定义控件的分类

在.NET 中，除了使用微软提供的 Windows 控件外，开发人员还需要开发自定义控件。开发人员可以确定自定义控件的布局、外观、绘制、用户响应的更多细节，从而开发出满足特定需要的控件。在.NET 中，通常可以实现 3 种自定义控件：扩展控件、复合控件和自定义控件。

- 扩展控件：扩展控件是指从现有的微软提供的.NET 类库控件中，直接继承和扩展而得的窗体控件。通过这种方法创建的窗体控件，既保留了 Windows 窗体控件的原有功能，又通过添加自定义属性、方法和事件等方式为新控件增加新的功能。例如，接下来在 7.2.3 节要创建的 HexTextBox 控件。
- 用户控件：.NET 类库提供 UserControl 类，它表示一个控件的公共容器，通过窗体设计器，可以为它添加一个或多个现有控件，将这些控件作为整体进行访问。用户控件就是指继承至 UserControl 类，又根据需要添加特定成员的用户控件，例如接下来在 7.2.4 节要创建的 Caculator 控件。
- 自定义控件：自定义控件是指直接从 Control 继承从头开始创建的控件。与扩展控件和用户控件相比，由于大量的实现都留给开发人员进行，因此创建自定义控件需要耗费更多的心思和精力，但自定义控件可以具有更大的灵活性，可以开发出更具个性和专用的用户界面。在应用软件开发中很少使用。

上面介绍的 3 种自定义控件开发方式各有优缺点，通常需要根据实际的需求进行选择。一般来说，如果要将若干个现有窗体控件的功能合成一个可作为整体使用的控件单元，那么从 UserControl 类派生用户控件会更加适合。如果大多数所需的功能已经与现有的窗体控件相同，只是简单增减一些后台功能，则通常使用扩展控件的方法。相反，如果想要提供控件的自定义图形化表示形式，需要实现无法从标准控件获得的自定义功能，可以考虑直接从 Control 类继承自定义控件。

7.2.2　创建.NET 控件库 MyControls

在 Visual Studio 2010 中提供一种.NET 类库模板——Windows 窗体控件库，专门用来开发自定义控件类库，实际上它和普通类库并没有本质上的区别，只是自动完成以下两个操作：

- 自动引用 Windows 窗体和控件开发时必需的引用 System.Drawing 和 System.Windows.Forms。
- 自动创建一个从 UserContorl 类继承的 UserControl1，而不是创建类 class1。

本节将介绍如何创建自定义控件库 MyControls，在 Visual Studio 2010 中，要创建一个 MyControls 需要如下两个步骤：

（1）打开 Visual Studio 2010 开发环境，通过“开始”|“新建”|“项目”菜单命令打开“新建项目”对话框。

（2）在 C#项目的“模板”列表中选择“Windows 窗体控件库”，在名称文本框中输入 MyControls。单击“确定”按钮创建 MyControls 类库项目。



图 7-3　MyControls 代码结构

Visual Studio 2010 自动生成的 MyControls 类库的代码结构，如图 7-3 所示。其中，System.Drawing 和 System.Windows.Forms 都是自动添加的引用，UserControl1 是自动添加的用户控件。

在“解决方案资源管理器”中右击项目 MyControls，在弹出的快捷菜单中选择“生成”或“重新生成”选项，生成项目，就可以得到 MyControls.dll，它包含了一个自定义控件 UserControl1。

7.2.3　继承 TextBox 实现十六进制数字控件 HexTextBox

假设，在某个应用软件中，要实现一个 HexTextBox 控件，它的主要功能是：

- 只能输入 32 位（即：4 字节）的十六进制无符号整数。
- 可以设置或获取文本框当前数值。
- 可以获取文本框当前数值的十六进制文本字符串。

基于这样一个控件需求，首先来看看现有的控件中是否存在符合要求或功能相近的控件，最终发现文本输入框 TextBox 与 HexTextBox 的功能很接近，所以直接从 TextBox 控件继承得到 HexTextBox 整数输入框。

在 Visual Studio 2010 中没有为扩展控件提供专门的模板，要从 TextBox 控件继承得到 HexTextBox 控件只能先创建普通类，然后手动修改一些代码来完成。在项目 MyControls 的基础上，通过以下 5 个步骤来完成 HexTextBox 控件的创建。

（1）右击项目 MyControls，在弹出的快捷菜单中选择“添加”|“类”命令。在弹出的“添加新项”对话框中，输入新建类的名称——HexTextBox。

（2）在代码编辑器中打开 HexTextBox 类的代码文件 HexTextBox.cs。

（3）在 HexTextBox.cs 中引用命名空间 System.Windows.Forms，因为 HexTextBox 的

基类 TextBoxs 属于该命名空间。

（4）将 HexTextBox 类修改成从 TextBox 类继承，为其添加默认构造函数，并用 Public 关键字将它公开给 MyControls 的调用者。

（5）为 HexTextBox 类添加表 7-1 所示的属性和方法，分别完成前面提出的功能。

表 7-1　HexTextBox主要成员

成员名称	可访问性	功 能 说 明
HexValue	Public	读写属性，uint 类型，获取或设置当前文本框中的十六进制的无符号整数值
HexString	Public	只读属性，string 类型，获取当前文本框中的数值的十六进制表示的字符串
OnKeyPress	Protected	重载方法，拦截用户按键操作，检查是否为合法的十六进制字符或者控制字符，如果合法则处理，否则不处理
IsValidChar	Private	新增方法，判断指定字符是否为合法的十六进制字符或者控制字符，如果是返回 true，否则返回 false

通过上面 5 个步骤，得到 HexTextBox 类的基本框架，此时从“解决方案资源管理器”中可以看到 HexTextBox 类的图标变成，表示它不是一个普通类，而是一个组件。示例代码 7-3 是 HexTextBox 控件的具体实现。

示例代码 7-3

```
using System.Windows.Forms;              //TextBox 类需要引用该命名空间
namespace MyControls
{
    public class HexTextBox : TextBox
    {
        //公开默认构造函数，直接调用 TextBox 的构造函数
        public HexTextBox( )
            : base( )
        { }
        //判断一个字符是否为合法的十六进制字符
        private bool IsValidChar(char ch)
        {
            if (char.IsDigit(ch))         //如果在“0~9”之间，则合法
            {
                return true;
            }
            if (char.IsControl(ch))       //如果是控制类字符，左移右移等，合法
            {
                return true;
            }
            ch = char.ToUpper(ch);        //变成大写，将'a'-'f'变成'A'-'F'
            if ((ch <= 'F') && (ch >= 'A'))
            {
                return true;
            }
            return false;                 //其他为不合法的字符
        }
        //重写 OnKeyPress()方法，首先判断输入的字符是否为十六进制字符
        //如果不是，则取消按键响应，否则调用 TextBox.OnKeyPress()处理
        protected override void OnKeyPress(KeyPressEventArgs e)
        {
            //判断输入字符是否合法
            if (!this.IsValidChar(e.KeyChar))
```

```
            {
                //不合法，则标记为已处理，则放弃数据的输入
                e.Handled = true;
            }
            e.KeyChar = char.ToUpper(e.KeyChar);     //将'a'-'f'变成'A'-'F'
            base.OnKeyPress(e);
        }
        //设置当前数值编辑框中的十六进制值
        public uint HexValue
        {
            get
            {
                //试图解析文本框中的字符串，如果是一个数字则返回，否则返回 0
                try
                {
                    uint val = uint.Parse(this.Text);
                    return val;
                }
                catch (Exception)
                {
                    return 0;
                }
            }
            set
            {
                //将要设置的数值转换成十六进制的字符串，并设置为文本框的值
                this.Text = value.ToString("X");
            }
        }
        //只读属性，获取当前文本框中输入数值的 8 字符十六进制字符串格式
        public string HexString
        {
            get
            {
                uint val = this.HexValue;                  //获取当前文本框的值
                return string.Format("{0:X8}", val);       //将值转换成字符串
            }
        }
    }
}
```

7.2.4　继承 UserControl 实现计算器控件 CaculatorUC

本节将绍如何通过继承 UserControl 类来实现用户控件，假设现在需要实现一个自定义控件 CaculatorUC，它具有以下功能：

- ❑ 具有一个输入框，用来接收操作数、整数或小数。
- ❑ 能执行加、减、乘、除等多种运算。
- ❑ 在每次运算完成时引发一个 CaculateDown 事件。

从整体上看，CaculatorUC 控件需要多个控件的组合，它需要 NumericUpDown 控件进行数值输入，需要按钮进行加、减、乘、除四种运算等。所以 CaculatorUC 很显然需要从 UserControl 类继承，而且组合多个现有控件来实现。在项目 MyControls 的基础上，需要以下 4 个步骤完成 CaculatorUC。

（1）通过向导新建一个用户控件，命名为 CaculatorUC。

（2）在窗体设计器中打开 CaculatorUC 控件，分别命名为 nudNumber 和 nudResult，前者输入数字，后者显示结果，且 nudResult 的 ReadOnly 属性为只读。

（3）添加 5 个 Button 控件到 CaculatorUC 控件上，分别实现加、减、乘、除、清除结果。

（4）添加表 7-2 所示的成员属性和方法，分别实现不同的功能。

表 7-2　CaculatorUC主要成员

成 员 名 称	可访问性	功 能 说 明
CaculateDown	Public	CaculateDown 事件，在每次计算完成之后都会引发该事件
OnCaculateDown	Protected	受保护的方法，用来引发 CaculateDown 事件
Result	Public	只读属性，decimal 类型，获取 CaculatorUC 当前的计算结果
btnClr_Click	Private	CLR 按钮的 Click 事件处理函数，清空 CaculatorUC 当前的计算结果
btnAdd_Click	Private	“+”按钮的 Click 事件处理函数，执行加法运算，并上报 CaculateDown 事件
btnSub_Click	Private	“–”按钮的 Click 事件处理函数，执行减法运算，并上报 CaculateDown 事件
btnMulti_Click	Private	“*”按钮的 Click 事件处理函数，执行乘法运算，并上报 CaculateDown 事件
btnDiv_Click	Private	“/”按钮的 Click 事件处理函数，执行除法运算，并上报 CaculateDown 事件

经过上面 4 个步骤，可以得到 CaculatorUC 控件的设计效果如图 7-4 所示，这样就得到了该控件的基本界面布局，接下来就要进一步实现它的后台数据处理。

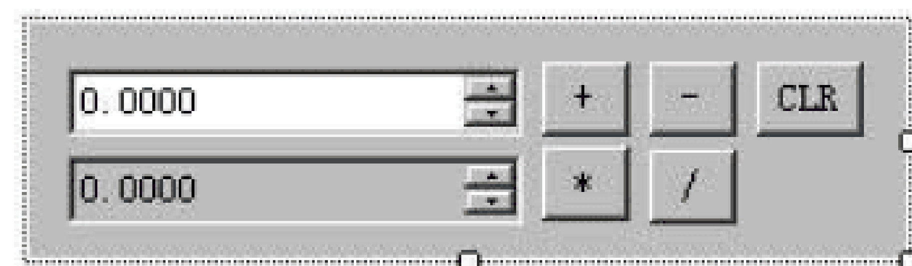


图 7-4　CaculatorUC 设计图

示例代码 7-4 是用户空间 CaculatorUC 的具体实现，从中可以看出自定义控件不仅可以包含字段、属性，同样可以包含事件等任何类型的成员，因为它本质上也是类。

示例代码 7-4

```
using System.Windows.Forms;
namespace MyControls
{
    public partial class CaculatorUC : UserControl //从 UserControl 类继承
    {
        public CaculatorUC( )
        {
            InitializeComponent( );
        }
        //CaculateDown 事件的定义和引发事件代码，为了简单，这里并没有太多事件参数
        public event EventHandler CaculateDown;
        protected void OnCaculateDown(EventArgs arg)
        {
            if (this.CaculateDown != null)
            {
                this.CaculateDown.Invoke(this, arg); //引发 CaculateDown 事件
            }
        }

        //只读属性，用来获取当前计算器中产生的结果
```

```
public decimal Result
{
    get
    {
        return this.nudResult.Value;           //从界面获取计算结果
    }
}

private void btnClr_Click(object sender, EventArgs e)
{
    this.nudNumber.Value = 0;                  //清空现有数据
    this.nudResult.Value = 0;
}
//清空界面上的数据
private void CaculatorUC_Load(object sender, EventArgs e)
{
    this.nudNumber.Value = 0;                  //清空现有数据
    this.nudResult.Value = 0;
}

//执行加法运算，并产生 CaculateDown 事件
private void btnAdd_Click(object sender, EventArgs e)
{
    decimal res = this.nudResult.Value;        //获取当前界面上的数据
    decimal val = this.nudNumber.Value;
    res = res + val;                           //执行加法运算
    this.nudResult.Value = res;                //将结果显示到界面
    this.nudNumber.Value = 0;                  //清空输入数据
    this.OnCaculateDown(null);                 //上报计算完成事件
}
//执行减法运算，并产生 CaculateDown 事件
private void btnSub Click(object sender, EventArgs e)
{
    decimal res = this.nudResult.Value;        //获取当前界面上的数据
    decimal val = this.nudNumber.Value;
    res = res - val;                           //执行减法运算
    this.nudResult.Value = res;                //将结果显示到界面
    this.nudNumber.Value = 0;                  //清空输入数据
    this.OnCaculateDown(null);                 //上报计算完成事件
}
//执行乘法运算，并产生 CaculateDown 事件
private void btnMulti_Click(object sender, EventArgs e)
{
    decimal res = this.nudResult.Value;        //获取当前界面上的数据
    decimal val = this.nudNumber.Value;
    res = res * val;                           //执行乘法运算
    this.nudResult.Value = res;                //将结果显示到界面
    this.nudNumber.Value = 0;                  //清空输入数据
    this.OnCaculateDown(null);                 //上报计算完成事件
}
//执行除法运算，并产生 CaculateDown 事件
private void btnDiv Click(object sender, EventArgs e)
{
    decimal res = this.nudResult.Value;        //获取当前界面上的数据
    decimal val = this.nudNumber.Value;
    res = res / val;                           //执行除法运算
    this.nudResult.Value = res;                //将结果显示到界面
```

```
            this.nudNumber.Value = 0;                    /清空输入数据
            this.OnCaculateDown(null);                   //上报计算完成事件
        }
    }
}
```

7.2.5　使用自定义控件 CaculatorUC

在自定义控件创建完成后，就可以在 Windows 窗体设计或其他自定义控件的设计和开发中使用它们。通常在使用自定义控件之前，需要先将它们添加到工具箱中，这样就可以像使用一般 Windows 控件那样进行可见即所得的设计。在 Visual Studio 2010 中，要将自定义控件添加到工具箱通常需要如下 4 个步骤。

（1）打开 Visual Studio 2010 的工具箱视图，然后打开工具箱的某一栏，这里选择“常规”栏。

（2）在右键快捷菜单中选择“选择项”选项，打开“选择工具箱项”对话框，如图 7-5 所示，在其中选中“.NET Framework 组件”选项卡。

（3）单击“浏览”按钮，在打开的文件对话框中打开要加载的自定义控件所在的 DLL，这里打开 7.4 节开发的 MyControls.dll。返回到“选择工具箱项”对话框，会自动选中可以加载的控件，如图 7-5 中自动选中了 CaculatorUC 控件。

（4）单击“确定”按钮，将选中的控件加载到“工具箱”中的当前打开栏中，如图 7-6 所示。这样在进行界面设计时，就可以直接从工具箱将这些自定义控件拖放到窗体设计器上。



图 7-5　选择工具箱项对话框



图 7-6　选择工具箱项对话框

自定义控件新增的 public 属性和事件都可以在“属性管理器”窗口中进行修改，本例通过实例 UseMyControls 介绍如何直接使用控件 CaculatorUC。在 Visual Studio 2010 中，通过如下 4 步创建该实例：

（1）打开 Visual Studio 2010，并创建名为 UseMyControls 的 Windows 窗体应用程序。

（2）在窗体设计器中打开窗体 Form1，并从工具箱中拖曳 1 个 CaculatorUC 控件和 1 个 Label 控件到 Form1 上。CaculatorUC 控件名为 CaculatorUC1，Label 控件名为 lbHint。

（3）通过属性管理器绑定 CaculatorUC1 的 CacluateDown 事件，默认为 CaculatorUC1_

CacluateDown()。并添加函数代码，从 CaculatorUC1 中获取 Result 并提示到 lbHint 上。示例代码 7-5 是实例 UseMyControls 的具体实现。

示例代码 7-5

```
namespace UseMyControls
{
    public partial class Form1 : Form
    {
        public Form1( )
        {
            InitializeComponent( );
        }
        private void caculatorUC1 CaculateDown(object sender, EventArgs e)
        {
            //从 CaculatorUC1 获取 Result 属性，并显示到 lbHint 上
            this.lbHint.Text = string.Format("The Result is {0}",
            this.caculatorUC1.Result);
        }
    }
}
```

图 7-7 为实例 UseMyControls 的运行效果图，从中可以看出，即使示例代码 7-5 只有简单的一个函数，但是由于自定义控件 CaculatorUC 实现了足够灵活的功能，所以该应用程序仍然可以实现强大的功能。

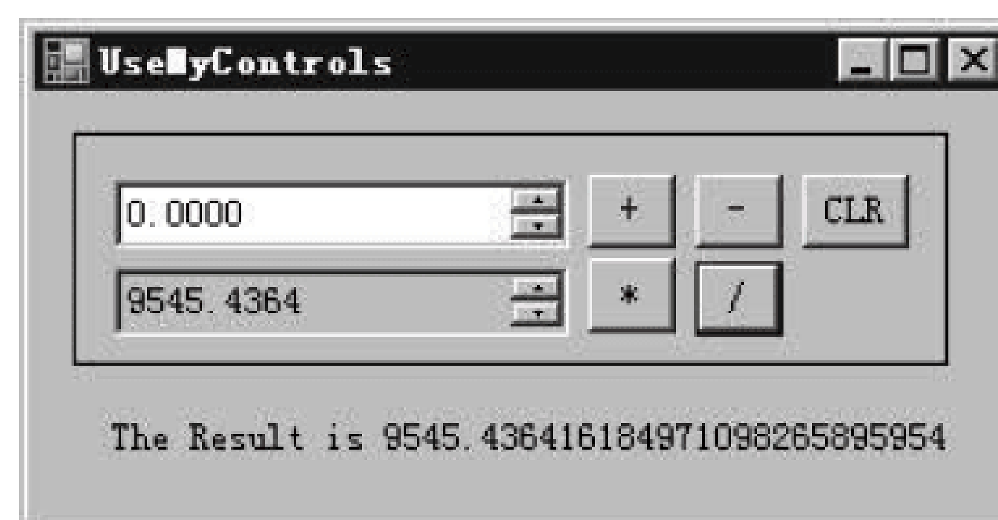


图 7-7　UseMyControls 效果图

注意：自定义控件的属性和事件，只有 public 修饰的才会公开到 Visual Studio 2010 的“属性管理器”中，并且可以对它们进行可见即所得的编辑。合理使用自定义控件，可以尽可能多的代码重用。

7.3　小　　结

.NET 类库是基于.NET 框架开发出来的动态链接库，它是应用软件模块划分和重用的基础，也是实际应用软件开发中最基本的技术。.NET 类库可以是普通的类型和接口集合，也可以是 Windows 控件集合，这些都可根据实际应用需要进行调整。

本章通过.NET 类库为主要目的，介绍如何通过 Visual Studio 2010 创建.NET 类库应用程序，以及自定义控件的开发。通过本章的学习，读者应该掌握以下知识点：

- Visual Studio 2010 提供哪几种.NET 类库应用程序模板？
- 在创建.NET 类库之前，应该注意哪些问题？
- 如何创建、生成和使用.NET 类库？
- 自定义控件有哪些类型？
- 如何创建扩展自定义控件？
- 如何创建用户自定义控件？
- 如何将自定义控件从.NET 类库加载到“工具箱”？
- 如何使用.NET 开发的自定义控件？

第 8 章　ASP.NET 网页开发

随着 Internet 的快速发展，网络在人们的日常生活中占据着越来越重要的地位。各种网站也成为了主要的信息来源，随着 Web 2.0 概念的提出及 XML 标准的普及，网站开发逐步发展成软件开发的一个重要部分。随着.NET 框架的推出，微软在发布 C#的同时，也在网页开发技术上对 ASP 进行了改进，发布新一代网页开发技术——ASP.NET。本章将简单介绍在.NET 框架下的网页开发技术。

8.1　ASP.NET 入门

ASP.NET 和 ASP 并没有本质的联系，它完全是新的开发技术，也逐渐取代了 ASP，并且可以与 C#等.NET 开发语言结合，使得网站的应用逻辑实现更加容易，也是的网站开发更加模块化。本节将介绍 ASP.NET 的基本概念。

8.1.1　什么是 ASP.NET

随着.NET 框架的推出，微软对原有的网页开发语言 ASP 进行改革，推出性能更好、使用更加方便的网页开发语言 ASP.NET。ASP.NET 本质上和 ASP 并没有联系，更不是 ASP 简单的版本改进，它和 ASP 有着完全不同的结构和开发模式，ASP.NET 使得开发网页可以更加快捷、更加模块化。

ASP.NET 作为.NET 框架的一个独立组件存在，提供了一个统一的 Web 开发模型。ASP.NET 建立在.NET 公共语言运行库上的编程框架，包括 Web 应用程序开发所必需的各种服务，可用于在服务器上生成功能强大的 Web 应用程序。

另外，ASP.NET 与.NET 框架无缝集成，在开发 Web 应用程序时，可以访问.NET 类库中的类和接口，也可以访问使用 C#、VB.NET 等语言开发的自定义类库（DLL 文件）。在开发过程中使用 C#、VB.NET 等开发语言，可以充分利用公共语言运行库、类型安全、继承等多种语言特性。与以前的 Web 开发模型（比如 ASP）相比，ASP.NET 提供了以下几个重要的优点。

- 增强的性能：ASP.NET 是在服务器上运行的编译好的公共语言运行库代码。与早期的 Web 开发模型不同，ASP.NET 可利用早期绑定、实时编译、本机优化等技术，相当于在编写代码行之前便显著提高了性能。
- 开发工具：Visual Studio 2010 为 ASP.NET 的开发提供了工具箱和设计器，可以通过控件拖放、属性设置等简单方式进行界面设计，后台通过 C#、VB.NET 等代码编辑器后台编码，使得开发 Web 应用程序方便快捷。

- 灵活性：ASP.NET 和.NET 框架集成，因此 Web 应用程序开发人员可以利用整个.NET 框架丰富的类库、消息处理和数据访问等技术。而且 ASP.NET 与语言无关，所以可以选择最适合应用程序的语言，或跨多种语言分割应用程序。
- 扩展性：ASP.NET 中，可以编写自定义组件扩展或替换 ASP.NET 运行库的任何子组件。
- 模块化：ASP.NET 使用前后台开发模式，后台通过 C#等.NET 编程语言开发网站处理逻辑，使得网站更加模块化，便于设计和管理。

和 Windows 窗体应用程序一样，在.NET 类库中也为 Web 应用程序提供了大量的类和接口，它们都封装在.NET 类库的 System.Web 命名空间下。System.Web 命名空间提供可以进行浏览器与服务器通信的类和接口，主要包括以下几项。

- HttpRequest 类：用于提供有关当前 HTTP 请求的广泛信息。
- HttpResponse 类：用于管理对客户端的 HTTP 输出。
- HttpServerUtility 类：用于提供对服务器端实用工具与进程的访问。
- 其他：还包括用于 Cookie 操作、文件传输、异常信息和输出缓存控制的类。

通过这些类，可以对整个网页的具体信息、当前连接信息、浏览器的 Cookie 情况等进行管理，还可以控制页面的缓存等。

除了上面这些和数据处理相关的类，Web 应用程序开发还需要一个重要的命名空间 System.Web.UI，它提供的类和接口可以用来创建在网页中作为用户界面元素的 ASP.NET 服务器控件和网页。System.Web.UI.WebControls 命名空间为 Web 开发人员提供大量常用的网页界面元素，如按钮、文本框、菜单、列表等，还包括日历等具有特殊用途的控件。

System.Web.UI.WebControls 命名空间下提供的 Web 服务器控件，虽然最后都表现为 HTML 标记语言，但由于它们运行在服务器上，因此可以以编程方式控制这些元素。其中，WebControl 类用作这些服务器控件类的基类，为它们提供最基本的实现。

8.1.2　创建 Web 网站应用程序

通过 Visual Studio 2010 开发环境，开发人员在开发 Web 应用程序时，默认是以前后台模式完成的，这种模式也是最简单和方便的。所谓的前后台模式是指在 Web 应用程序开发时，前台的网页界面是一个以 aspx 为后缀的脚本文件，该脚本文件被集成在 IIS 中的 ASP.NET 所编译，然后被客户端访问。在 ASPX 文件后台，还包含一个具体的代码文件，C#语言则以 cs 为后缀，VB.NET 语言则以 vb 为后缀，后台代码被编译成.NET，网页根据需要访问对应的函数和方法。

在前后台开发模式中，前台的 aspx 文件只包含网页的界面部分，包括控件布局、颜色、样式、主题等，后台的代码文件（如 cs 文件）则负责逻辑功能的具体实现，比如访问数据库、数据运算等。前后台的开发模式的最大好处是界面和逻辑分离，界面设计人员和代码开发人员可以完全独立各自的工作，不必受彼此的约束。

Visual Studio 2010 提供了多个网站模板，如图 8-1 所示，其中最常用的是“ASP.NET 网站”。现在，通过 Visual Studio 2010 创建第一个 Web 应用程序，这是一个简单的网页，只有一个欢迎画面。需要以下 3 个步骤来完成。

（1）打开 Visual Studio 2010 开发环境，通过“文件”|“新建”|“网站”菜单，打开“新建网站”对话框，如图 8-1 所示。

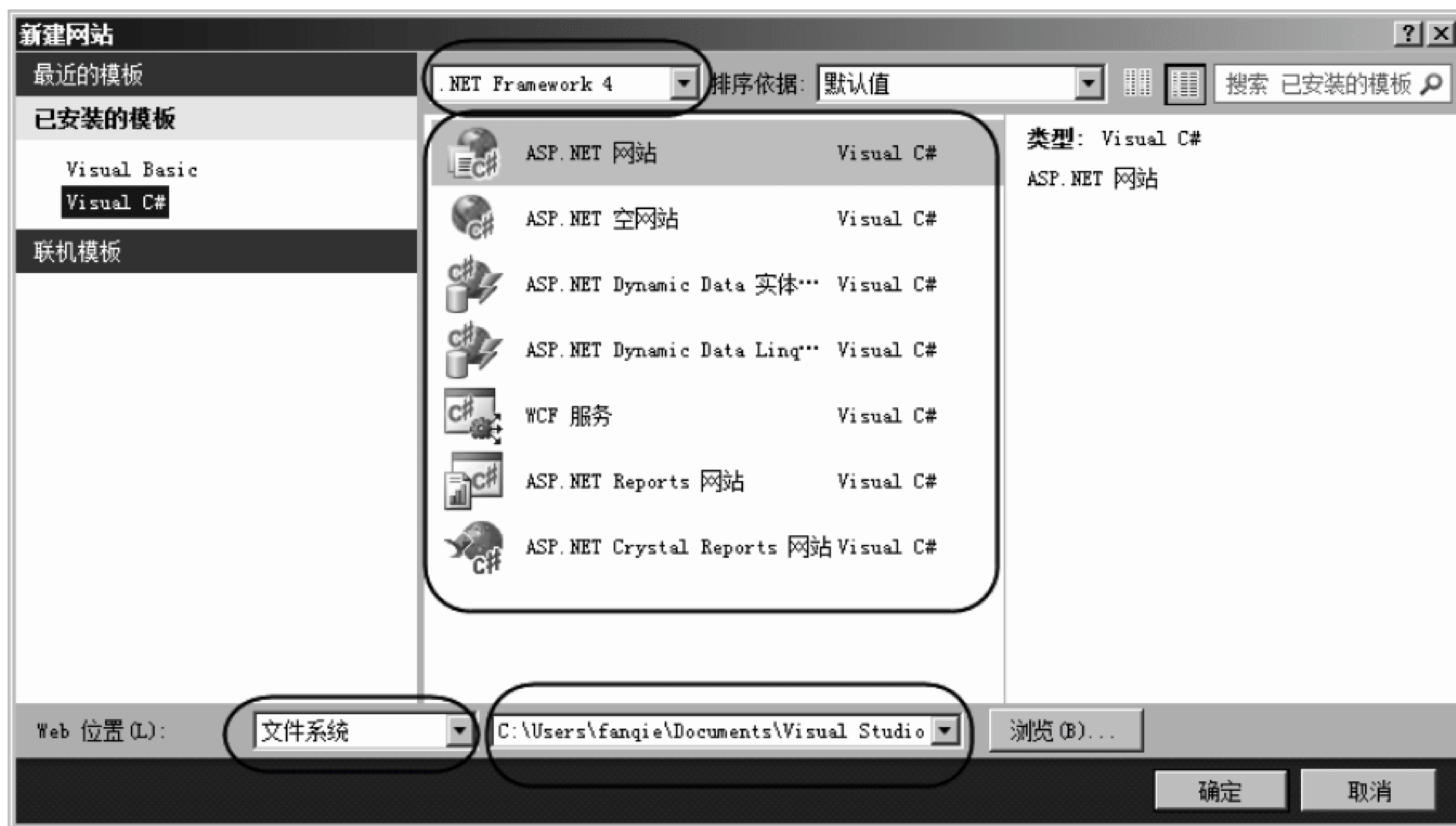


图 8-1　新建网站对话框

（2）在模板列表中选择“ASP.NET 网站”，在位置下拉框中选择“文件系统”，并浏览要保存网站的本地目录。“语言”列表中选择后台代码的语言，这里选择 Visual C#。

（3）单击“确定”按钮完成新网站的创建。

注意：当网站的位置类型为“文件系统”时，网站所有的相关文件（网页、图片等）都保存在这个目录下，所以目录名称通常为网站的名称。本例使用 WelcomeSite。

网站 WelcomeSite 成功创建后，Visual Studio 2010 会自动生成一个默认的网页——Default，网页界面文件为 Default.aspx，该网页还是空白，其中代码文件 Default.aspx.cs 是网页 Default 的后台实现。

8.1.3　通过网页设计器编辑 Web 网页

网站 WelcomeSite 创建完成后，Default.aspx 默认没有任何内容，需要通过 Visual Studio 2010 提供的 Web 窗体设计器进行网页设计和编码。首先，在“解决方案资源管理器”中双击“Default.aspx”可以在 Web 窗体设计器（如图 8-2 所示）中打该网页进行编辑，包含“设计”和“源代码”两种编辑模式。

在图 8-2 中，通过标记为 1 的按钮选择“设计”模式，该模式下所有编辑都是可视化的，而且所见即所得，可以直接将“工具箱”中的控件拖动到设计器中，通过“属性管理器”设置网页和控件的外观属性等。通过标记为 2 的按钮选择“源代码”模式，该模式是直接查看和修改 ASP.NET 语言代码。通过标记为 4 的按钮可以同时显示“设计”和“源代码”模式，但由于窗体会被拆分，所以可见区域减少。

值得注意的是，“设计”和“源代码”两种编辑模式是完全同步的，在任何一种模式

下修改了网页，都会反映到另外一种模式中。在 ASP.NET 网页中，每个界面元素都有一个路径，当前正在编辑的界面元素的路径会显示在图 8-2 中标记为 3 的位置，也可以在这里选择要编辑的元素。

在 WelcomeSite 实例中，首先用“设计”模式进行编辑，并在界面上直接输入一段文本，通过“工具栏”编辑它的颜色和字体等属性（就像使用 Word 一样）。然后在“源”模式下将页面标题设置为“欢迎学习 ASP.NET 知识”。最后从“源”模式下看到的 ASP.NET 代码如示例代码 8-1 所示。

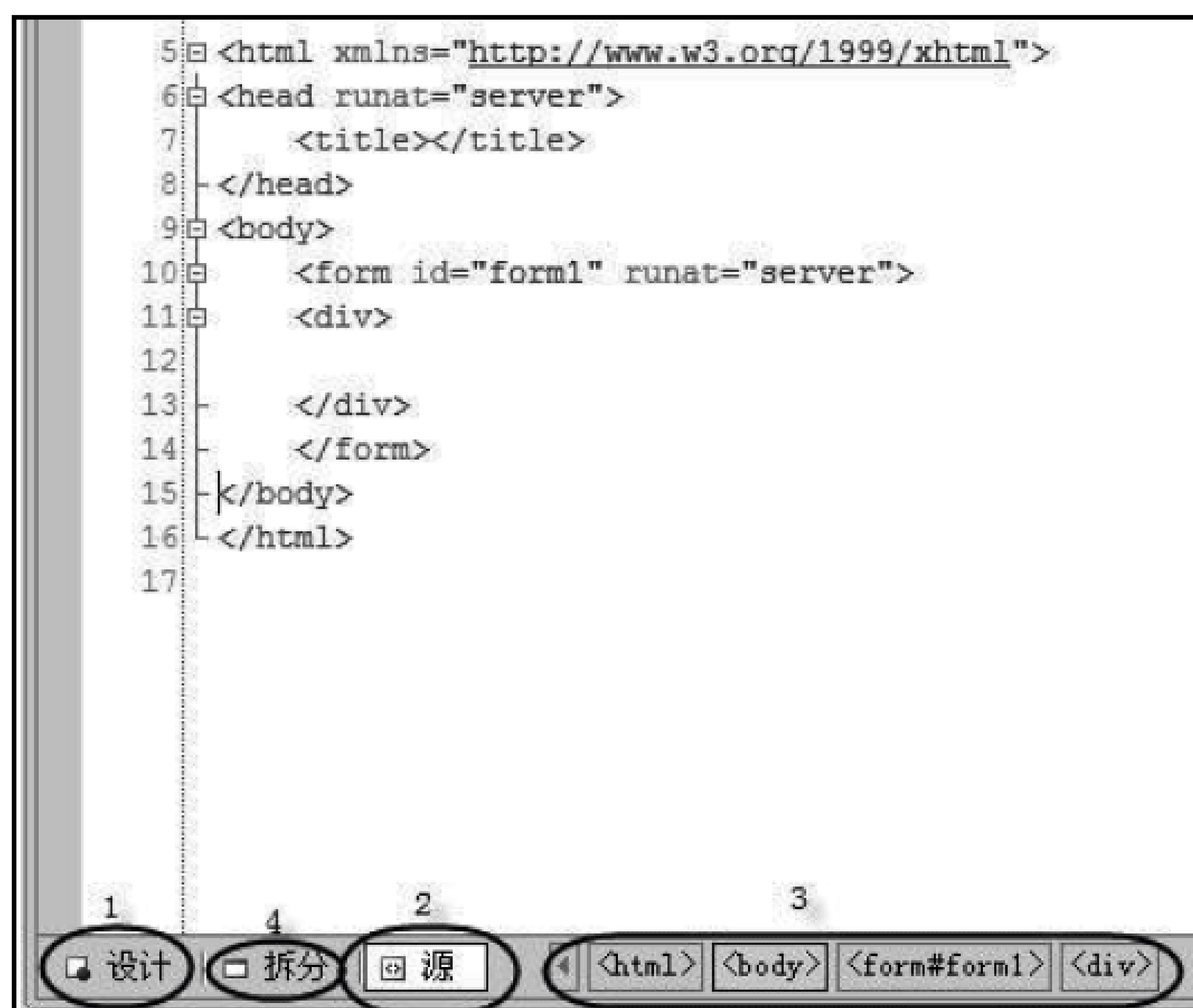


图 8-2　网页设计器源代码视图

示例代码 8-1

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits=" Default" %>
<!DOCTYPE  html  PUBLIC  "-//W3C//DTD  XHTML  1.0  Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>欢迎学习 ASP.NET 知识</title>
    <style type="text/css">
        .style1
        {
            color: #FF3300;
        }
        .style2
        {
            color: #000099;
            font-weight: bold;
        }
        .style3
        {
            color: #FF3300;
            background-color: #99CC00;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <span class="style1">您好，欢迎阅读本书！</span><br />
        这是本章介绍的第一个<span class="style2">ASP.NET</span>网站，接下来会有
<span class="style3">更多精彩的例子</span>等着你，<br />
        千万不要错过！<b>o(∩_∩)o...</b></div>
    </form>
</body>
</html>
```

示例代码 8-1 中，<%@ Page……%>结点是对整个页面属性的设置，这些属性可以通

过“属性管理器”设置。这里，Laguage 属性表示后台代码文件的语言，如 C#。AutoEvent Wireup 属性表示控件和页面的事件是否自动匹配，通常为 true。CodeFile 属性则表示后台代码文件名，如 Default.aspx.cs。“<head>……</head>”结点则定义网页的头部信息，其中 title 元素是网页的标题。“<body>……</body>”实现网页呈现给用户的具体内容。通过菜单“调试”|“启动调试”命令查看网页的效果，如图 8-3 所示。



图 8-3　WelcomSite 页面效果图

注意：调试 ASP.NET 网页，使用的是 Visual Studio 2010 内置的模拟 IIS 服务器，不需要单独安装 IIS。但是如果要发布网站，那么服务器必须安装 IIS 网络服务器组件，可以在 WindowsXP 的系统安装盘上找到。

8.2　使用 ASP.NET 控件

.NET 类库为网站开发人员提供了多种丰富的 Web 服务器控件，这些控件包括按钮、文本框、列表框等，本节将详细介绍其中最常用的几个 Web 服务器控件。

8.2.1　添加网页 MyPage 到 WelcomeSite

一个网站，通常包含多个网页，所以添加新网页到已有网站是最常用的操作之一。在 Visual Studio 2010 中，通过以下 3 个步骤添加新网页到网站，这里将新网页 MyPage 添加到网站 WelcomeSite 中。

（1）在 Visual Studio 2010 中打开网站 WelcomeSite，在“解决方案资源管理器”中选择 WelcomeSite 结点，在右键快捷菜单中选择“添加新项”选项，弹出如图 8-4 所示的“添加新项”对话框。

（2）在其中选择“Web 窗体”模板，在名称文本框中输入新网页名称 MyPage。在语言栏选择新网页后台编程语言，这里为 C#。

（3）单击“确定”按钮，添加新网页 MyPage 到网站 WelcomeSite 中。

注意：在 ASP.NET 中，一个网站中的多个网页理论上可以具有不同的后台开发语言，但是实际开发中尽量使用相同的后台开发语言，这样不仅代码可读性好，接口也容易确定。



图 8-4　添加新网页 MyPage 对话框

一个 ASP.NET 网页（Page）通常包含一个或多个 Form，Web 网页中的 Form 类似于 WinForm 窗体开发中的窗体 Form，它是通常作为 Web 元素的容器，本身也是网页中的一个控件。Form 窗体由脚本<asp:from>声明，属性 runat 的值为 server，表示该控件为服务器控件。根据需要如下脚本定义一个 Form 名为 form1，并且运行在服务器。另外，可以为网页设置背景色、前景色、字体、样式等外观属性，可以响应它的各种事件。

```
<form id="form1" runat="server" />
```

在 ASP.NET 中，任何一个网页都是一个从 System.Web.UI.Page 类派生而来的类，Page 类实现了网页工作的基本框架和功能，开发人员只实现应用需要的功能即可。所以，一个 ASP.NET 网页通常包含 3 个代码文件，如这里创建的网页 MyPage。

- *.aspx：这是 ASP.NET 脚本文件，它定义了网页的外观和布局，如 MyPage.aspx。
- *.aspx.cs：它和*.aspx 文件完全对应，实际是一个分部类，实现了网页的部分逻辑，如 MyPage.aspx.cs。注意，如果语言是 VB.NET，该文件会是 MyPage.aspx.vb。
- *.cs：它也是一个分部类，定义了网页的后台逻辑实现，如 MyPage.cs。

在编译和生成 ASP.NET 网页时，编译器会按照图 8-5 所示的流程进行。首先，将*.aspx 和*.aspx.cs 文件合并到一起得到表示网页界面的部分类（MyPage 类的一部分），然后再将它与表示网页逻辑的*.cs（MyPage 类的另一部分）一起得到完整的网页类（MyPage 类）。最终将 MyPage 根据访问者的需要，生成为 HTML 文本发送到访问者浏览器上。

注意： 网页最终都是要以 HTML 格式的文本发送到浏览器上的，浏览器再根据这些文本呈现网页到界面上。所以网页的大部分运算和逻辑处理都是在服务器端进行的，客户端只是进行简单的运算和验证。

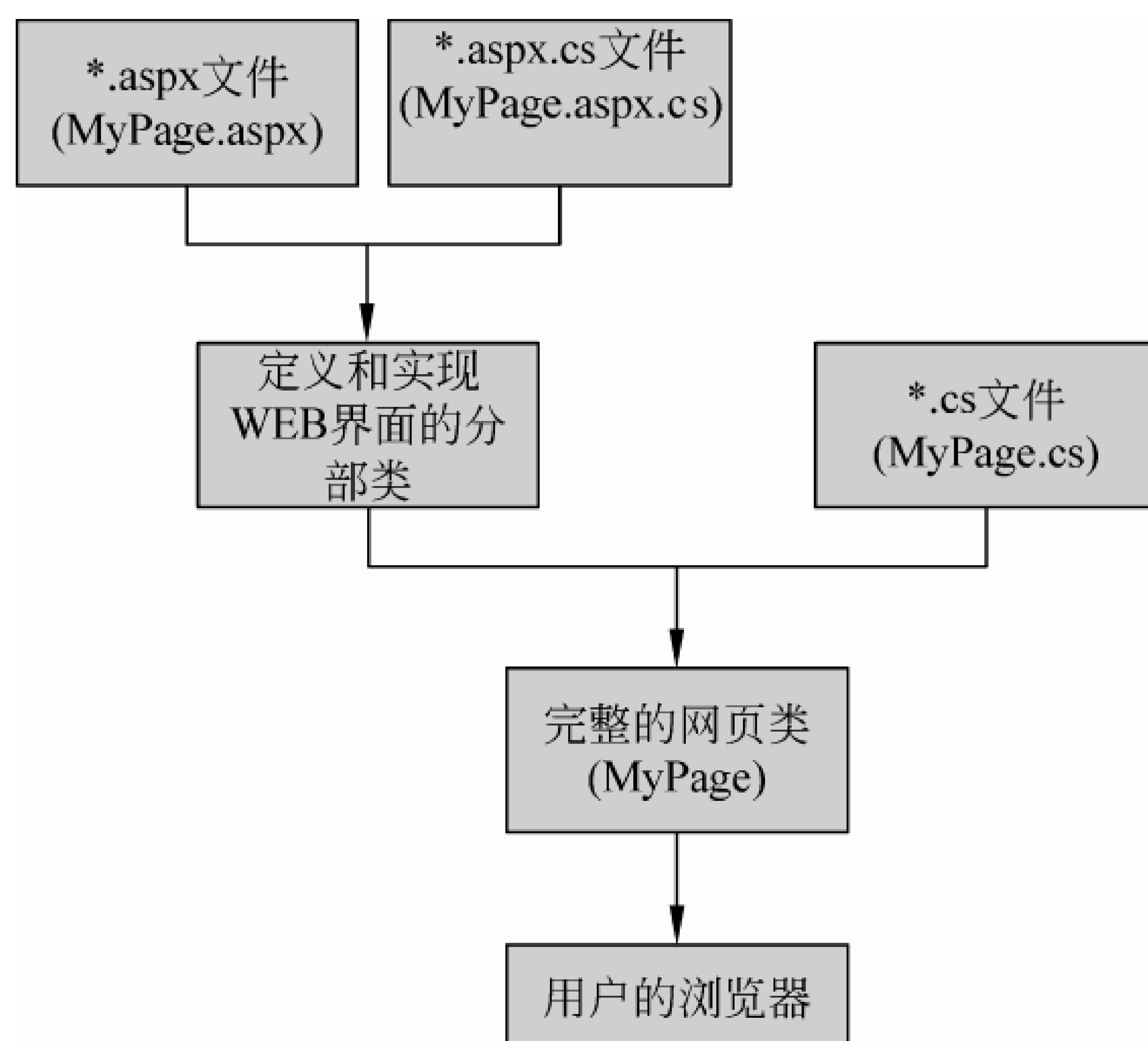


图 8-5　网页生成流程——MyPage 为例

8.2.2　用 TextBox 控件输入数据

在网页开发中，文本输入框控件是由 System.Web.UI.TextBox 类实现，为用户提供了一种在网页中输入数据（包括文本、数字和日期等）的方法。TextBox 服务器控件由脚本<asp:TextBox>声明，属性 runat 的值为 server，表示该控件为服务器控件，如下脚本表示控件 TextBox1 是服务器端运行的文本输入框控件。

```
<asp:TextBox ID="TextBox1" runat="server">文本框默认内容</asp:TextBox>
```

Web 窗体中的 TextBox 控件通过 TextMode 属性指定具体的输入模式，包括 3 种输入模式：单行（SingleLine）、密码（Password）和多行（MultiLine）。其中，SingleLine 模式只接收单行的文本输入，Password 模式也只能接收单行文本输入，但是输入字符用“*”隐藏，MultiLine 模式则允许用户输入多行文本。另外，TextBox 文本输入框还有字体和颜色、边框和背景等属性可以设置，这些属性都可以通过“属性管理器”来设置，也可以在 ASP.NET 脚本代码中编写。

示例代码 8-2 中，定义了 4 个文本输入框，它们都是单行输入模式，TextBox1 只是简单的单行文本输入框，TextBox2 则是红色且字体为楷体的文本输入框，TextBox3 是修改了边框样式和背景颜色的输入框，TextBox4 是一个 Enable 属性为 false 的不可用文本框，TextBox5 则是一个只读文本框。

示例代码 8-2

```
<head runat="server">
    <title>使用文本框—UserTextBox</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        默认文本框：<asp:TextBox ID="TextBox1" runat="server">默认文本框</asp:
        TextBox>
```

```
        <br />
        蓝色楷体文本框：<asp:TextBox ID="TextBox2" runat="server" Font-Names=
        "楷体_GB2312" ForeColor="Blue">红色楷体文本</asp:TextBox>
        <br />
        虚线边框文本框：<asp:TextBox ID="TextBox3" runat="server" BackColor=
        "#FFC0C0" BorderColor="#00C000" BorderStyle="Dashed" Font-Names="
        楷体_GB2312" ForeColor="Gray">虚线边框文本框</asp:TextBox>
        <br />
        不可用文本框：<asp:TextBox ID="TextBox4" runat="server" Enabled=
        "false">不可用文本框</asp:TextBox>
        <br />
        只读文本框：<asp:TextBox ID="TextBox5" runat="server" ReadOnly=
        "true">只读文本框</asp:TextBox>
    </div>
    </form>
</body>
</html>
```

示例代码 8-2 所得的网页，得到各种文本框的效果如图 8-6 所示。文本框控件的属性还有很多，读者可以通过“属性管理器”修改不同的属性得到各种外观。



图 8-6　文本框控件效果

8.2.3　用 Button 控件实现按钮

在 ASP.NET 网页中，可以使用 Button 和 LinkButton 控件实现按钮功能。Button 控件由类 System.Web.UI.Button 实现，在网页上由脚本<asp:Button>声明，可以为它配置不同的边框属性（宽度、类型、颜色）和文本属性（前景色、背景色、字体等）得到不同样式的风格。LinkButton 和 Button 控件一样，也可以用作一个普通的按钮，但是它以一个超链接的形式显示，由脚本<asp:LinkButton>声明。

作为按钮控件，Button 和 LinkButton 都包含 Click 事件，并将事件传回到服务器进行处理，通过 OnClick 属性来声明事件处理函数。下面代码就声明一个 Button 控件，名称（ID）为 Button1，显示文本（Text）为“按钮一”，Click 事件处理函数（OnClick）为 Button1_Click。这样当用户在网页上单击 Button1 按钮时，服务器会调用 Button1_Click 函数，对用户的请求进行处理。

```
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text="按
钮一" Width="113px" />
```

LinkButton 控件的声明和事件绑定与 Button 控件类似，设置控件属性和绑定事件都可以通过“属性管理器”的属性和事件两个面板可视化完成，Visual Studio 2010 会自动生成代码。

示例代码 8-3 的前半部分是网页的 ASP.NET 脚本部分，它声明了一个名为 btnAdd 的 Button 控件，并且设置了边框样式，字体和颜色等，还指定 Click 事件响应函数为 btnAdd_Click()。后半部分是用 C#语言实现的后台代码，本例中只有 btnAdd 的 Click 事件处理函数 btnAdd_Click()，它从界面的文本框中获取用户输入的数字，并进行加法运算，然后将计算结果显示到界面。

示例代码 8-3

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:TextBox ID="tbVal1" runat="server" Width="61px">12.35</asp:
        TextBox>
         + 
        <asp:TextBox ID="tbVal2" runat="server" Width="58px">35.11</asp:
        TextBox>
         
        <asp:Button ID="btnAdd" runat="server" Text=" = " BackColor=
        "#0033CC"
            BorderColor="#FF99CC" BorderStyle="Outset" BorderWidth="2px"
            Font-Bold="true"
            onclick="btnAdd_Click" />
         
        <asp:TextBox ID="tbResult" runat="server" Width="58px" ReadOnly=
        "true"></asp:TextBox>
    </div>
    </form>
</body>
public partial class UseButton : System.Web.UI.Page
{
    protected void btnAdd_Click(object sender, EventArgs e)
    {
        try
        {
            //从界面获取进行加法运算的两个参数 val1 和 val2
            double val1 = double.Parse(this.tbVal1.Text.Trim( ));
            double val2 = double.Parse(this.tbVal2.Text.Trim( ));
            //执行加法计算，并将结果显示到结果文本框 tbResult 上
            double result = val1 + val2;
            this.tbResult.Text = result.ToString( );
        }
        catch (System.Exception)
        {
            //如果发生异常，则在结果文本框中提示
            this.tbResult.Text = "Exception...";
        }
    }
}
```

生成并浏览该网页，得到如图 8-7 所示的网页效果，从中可以看出按钮的样式设置比较灵活，另外读者可以通过“属性管理器”修改不同的属性得到各种外观，比如为它指定图片等。

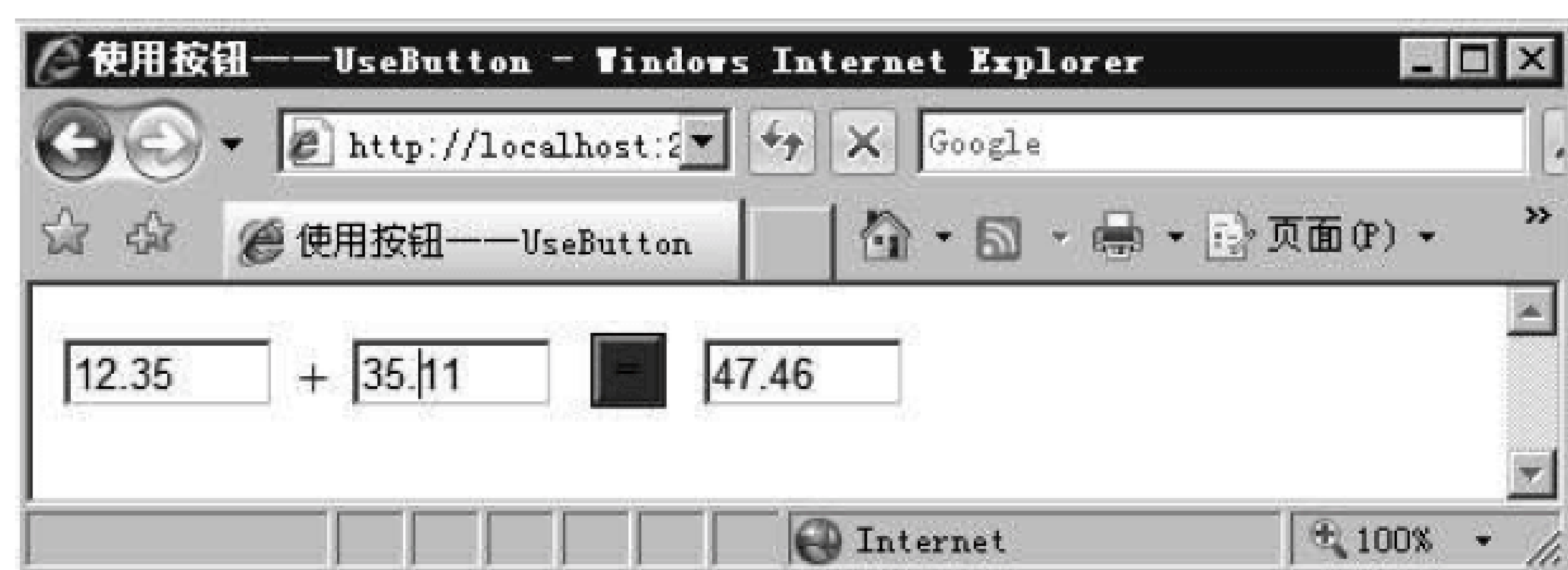


图 8-7　按钮控件效果图

8.2.4　用 HyperLink 控件实现超链接

在网站开发中，网页之间的跳转或是网页内部跳转都是常用功能。在 ASP.NET 中可以通过 HyperLink 控件实现超链接功能。除了可以设置 HyperLink 的字体、背景色、前景色等外观之外，HyperLink 包含两个主要属性。

- ❑ NavigateUrl：该属性表示要链接到的目标网页，可以是同一个网站内的页面，如前面的 Default.aspx，也可以是外部网站的页面，如 http://www.baidu.com。
- ❑ Target：表示目标网页的打开方式，包括以下 5 个可选值。
  - ➢ _blank：将内容呈现在一个没有框架的新窗口中。
  - ➢ _parent：将内容呈现在上一个框架的父级中。
  - ➢ _search：将在搜索窗体中呈现内容。
  - ➢ _self：将内容呈现在含焦点的框架中。
  - ➢ _top：将内容呈现在没有框架的全窗体中。

合理使用 Target 属性，可以让网站的使用变得简单方便，示例代码 8-4 中定义了 3 个 HyperLink 控件，分别浏览到前面几节实现的 3 个网页，从它们的 NavigateUrl 属性值可以看出，目标 URL 可以是相对的“~/Default.aspx”，当然也可以是绝对的。生成并运行该示例，可以看到当浏览到“使用 TextBox 控件”网页时，它会在新窗口中打开，因为它的 Target 属性是_blank。

示例代码 8-4

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>使用超链接——UseHypelink</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:HyperLink ID="hlink1" runat="server" NavigateUrl="~/Default.
        aspx" Width="93px">欢迎主页</asp:HyperLink>
        <br />
        <asp:HyperLink ID="hlink2" runat="server" NavigateUrl="~/UseButton.
        aspx" Target=" top"
            Width="123px">使用 Button 控件</asp:HyperLink>
        <br />
        <asp:HyperLink ID="hlink4" runat="server" NavigateUrl="~/UseText
        Box.aspx" Target="_blank"
            Width="123px">使用 TextBox 控件</asp:HyperLink>
```

```
    </div>
    </form>
</body>
</html>
```

8.2.5　用 DropDownList 和 ListBox 实现列表

在网页控件中用于列表显示数据的通常有 DropDownList 和 ListBox 两个控件。DropDownList 控件和 Form 窗体中的 ComboBox 控件类似，以下拉列表的方式显示可选择数据，并提供选中索引变化（SelectIndexChanged）事件。通过 DropDownList 控件的 Items 属性可以添加和删除列表中的元素，并且还可以将它的 Items 属性静态绑定到数据库等数据源。

网页上的 ListBox 控件是对多个可选项进行选择的最好方式，可以是单选（Single）也可以是多选（Multiple）。通过 ListBox 的 Items 属性也可以添加、移除列表中的可选项。当然，同样可以设置 ListBox 和 DropDownList 控件的字体、背景色、前景色、边框等外观样式。

示例代码 8-5 是实例 UseList 的代码，前半部分是页面的界面定义，它包括 1 个 DrowDownList 控件 cmbItems 和 1 个 ListBox 控件 lstSelected，前者表示所有的可选项，后者表示目前已经选中的项。后半部分主要是后台 C#实现的逻辑代码，主要是“添加”和“清除”两个按钮的 Click 事件处理函数。

示例代码 8-5

```
<head runat="server">
    <title>使用列表控件——UseList</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        欢迎来到 ASP.NET 学习园地，请选择您想学习的内容。<br />
        可选知识有：<br />
        <asp:DropDownList ID="cmbItems" runat="server" Height="20px" Width=
        "152px">
            <asp:ListItem>TextBox:文本框</asp:ListItem>
            <asp:ListItem>Button:按钮</asp:ListItem>
            <asp:ListItem>Label:标签</asp:ListItem>
            <asp:ListItem>HypeLink:超链接</asp:ListItem>
            <asp:ListItem>DrowpDownList:下拉框</asp:ListItem>
            <asp:ListItem>ListBox:列表框</asp:ListItem>
        </asp:DropDownList>
         
        <asp:Button ID="btnAdd" runat="server" Text="添加" onclick="btnAdd_
        Click" />
        <br />
        你已经选了：<br />
        <asp:ListBox ID="lstSelected" runat="server" Height="94px" Width=
        "148px">
        </asp:ListBox>
         
        <asp:Button ID="btnClr" runat="server" Text="清除" onclick="btnClr_
```

```
        Click"
          style="height: 26px" />
    </div>
    </form>
</body>
public partial class UseList : System.Web.UI.Page
{
    protected void Page Load(object sender, EventArgs e)
    {
        if(!this.IsPostBack)
        {
            this.cmbItems.SelectedIndex = 0;     //设置默认选中项为第一项
            //设置列表框允许多选
            this.lstSelected.SelectionMode = ListSelectionMode.Multiple;
        }
    }
    protected void btnAdd Click(object sender, EventArgs e)
    {
        //获取下拉列表框中选中的项，并添加到列表框中，可以重复添加
        this.lstSelected.Items.Add(this.cmbItems.SelectedItem);
    }
    protected void btnClr Click(object sender, EventArgs e)
    {
        this.lstSelected.Items.Clear( );         //清除列表框中的所有元素
    }
}
```

生成并浏览该网页，得到如图 8-8 所示的网页效果，从中可以看出 ASP.NET 里面的列表控件和 WinForm 窗体里的列表控件外观及功能都相似。

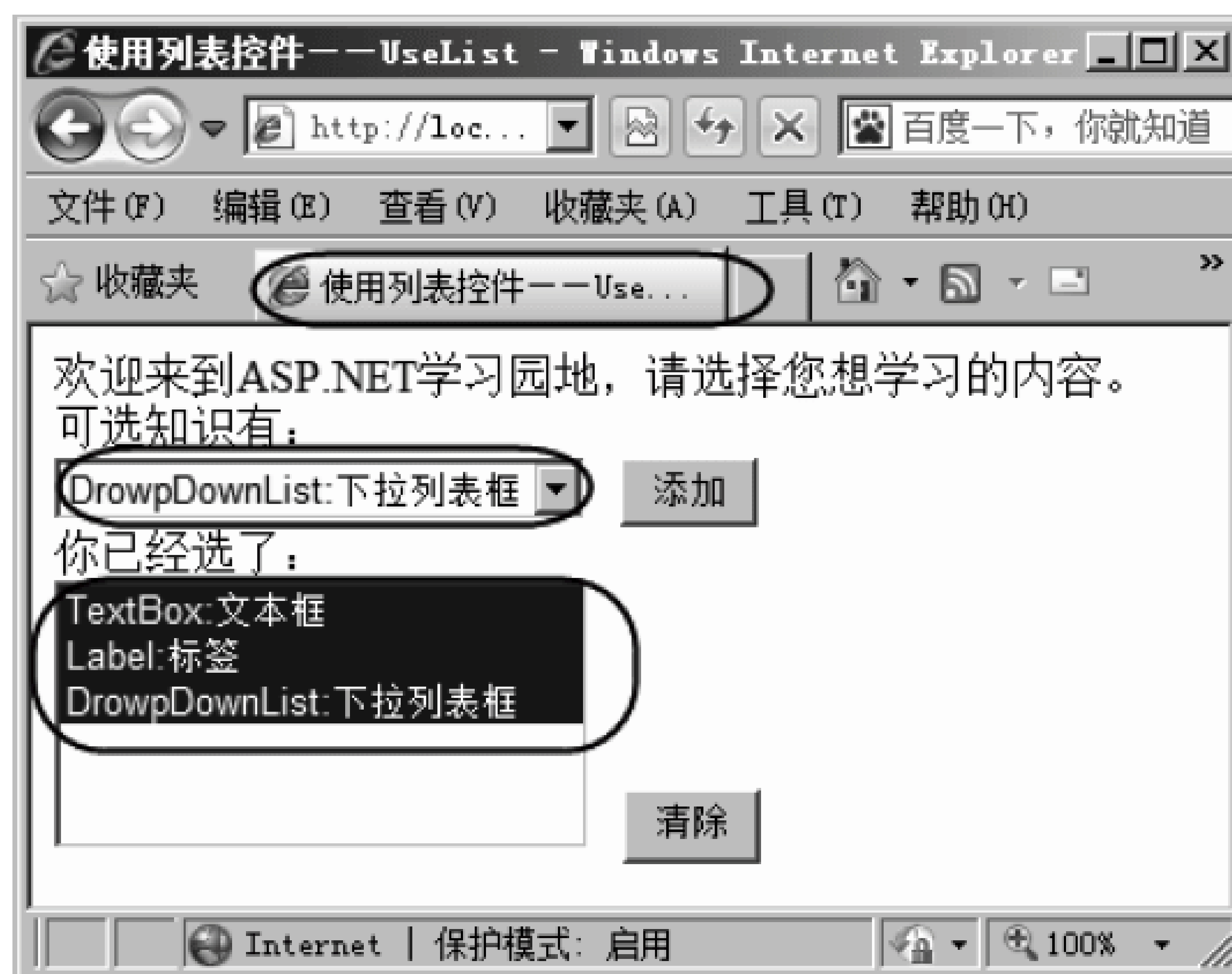


图 8-8　列表控件效果图

8.2.6　用 Menu 控件实现菜单导航

导航菜单是网站开发中的一个重要控件，它可以让网站结构更清晰，浏览者可以根据菜单查看感兴趣的网页等，对界面友好性有很大帮助。ASP.NET 中也提供了 Menu 控件实现网站上的菜单。

在 ASP.NET 中，可以为 Menu 控件设置不同的外观样式，可以修改它的背景色、前景色、字体等外观属性。一个菜单通常包含多个分层的菜单项，这些菜单项可以从一个数据源动态绑定，也可以通过菜单编辑器手动设置，如图 8-9 所示，每个菜单项可以设置它的可用性（Enable）、显示图片（ImageUrl）、目标网页（NavigateUrl）、弹出图片（PopOutImageUrl）、显示文本（Text）、目标网页打开方式（Target）等属性，合理地设置这些属性，尤其是显示图片等外观属性，可以让网页的菜单非常漂亮和友好。

另外，在 Web 页面中，菜单本身具有横排和竖排两种显示方向（Form 窗体只能横排），通过 Menu 控件的 Orientation 属性来设置它的显示方向，包括 Vertical 和 Horizontal 两个可选值。

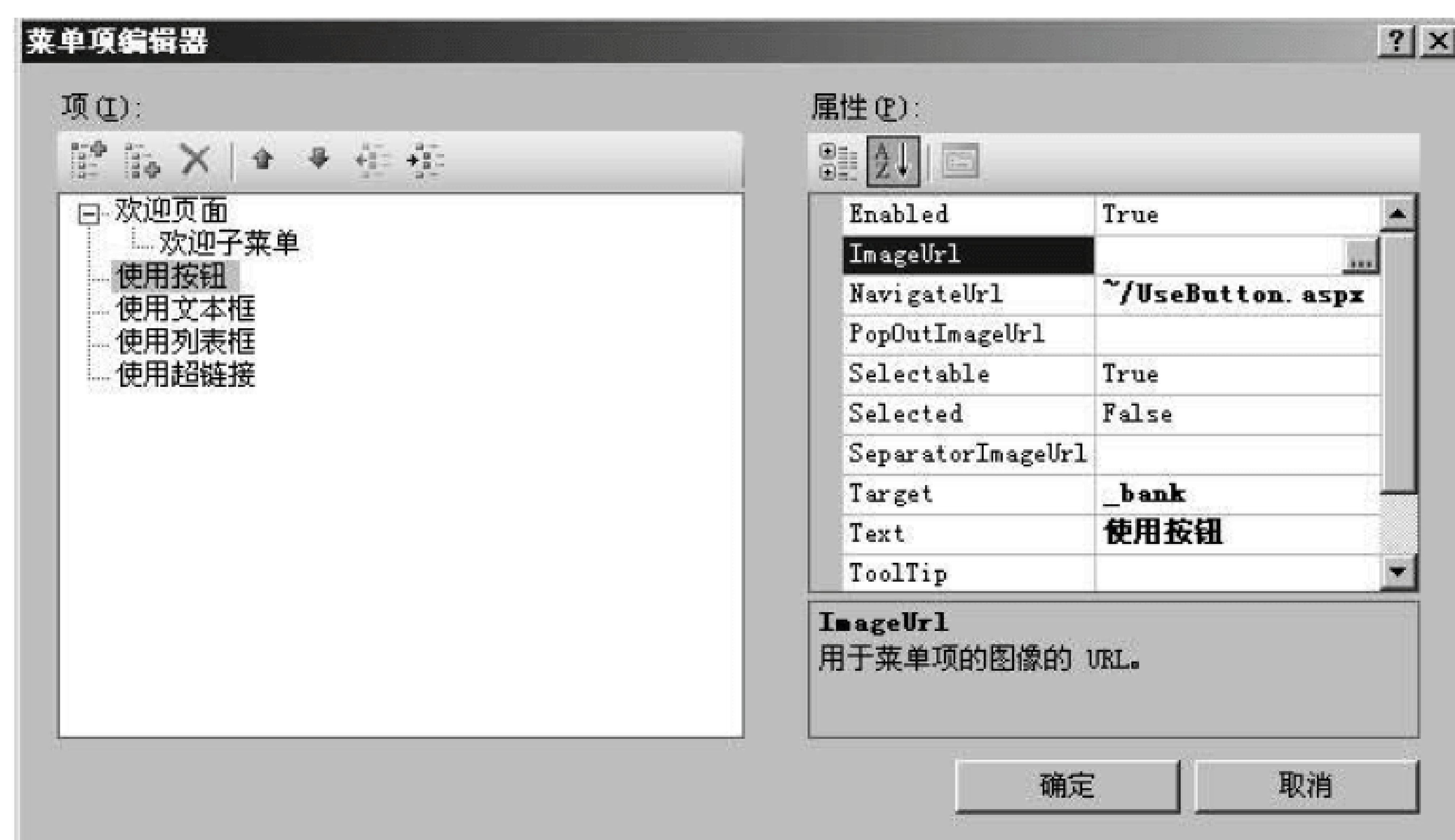


图 8-9　Web 菜单编辑器

示例代码 8-6 中，定义了一个 Menu 控件 Menu1，它用来导航前面几节创建的几个示例网页，从代码中 Menu1 的 Items 属性定义可以看出它包含 5 个一级菜单，其中子菜单“欢迎页面”又包含一个二级菜单“欢迎子菜单”。另外，再添加 1 个 Button 控件 Button1，在它的 Click 事件处理函数 Button1_Click()中实现菜单 Menu1 显示方向的自动切换。

示例代码 8-6

```
<head runat="server">
    <title>使用菜单</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        欢迎来到 ASP.NET 学习园地，下面是一个菜单控件示例。<br />
        这里的菜单采用了"自动套用格式"功能。<br />
        可以通过"改变方向"按钮来改变菜单排列的方向。<br />
        <asp:Menu ID="Menu1" runat="server" BackColor="#B5C7DE"
        …
            <Items>
                <asp:MenuItem NavigateUrl="~/Default.aspx" Target="_bank"
                Text="欢迎页面" Value="欢迎页面">
                    <asp:MenuItem Text="欢迎子菜单" Value="欢迎子菜单"></asp:
                    MenuItem>
                </asp:MenuItem>
                <asp:MenuItem NavigateUrl="~/UseButton.aspx" Target="_bank"
                Text="使用按钮" Value="使用按钮">
                </asp:MenuItem>
                <asp:MenuItem NavigateUrl="~/UseTextBox.aspx" Target=
                " bank" Text="使用文本框" Value="使用文本框">
                </asp:MenuItem>
                <asp:MenuItem NavigateUrl="~/UseList.aspx" Target="_bank"
                Text="使用列表框" Value="使用列表框">
                </asp:MenuItem>
                <asp:MenuItem Target=" bank" Text="使用超链接" Value="使用超链
                接"></asp:MenuItem>
            </Items>
        </asp:Menu>
```

```
    </div>
        <br />
        <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
        Text="改变方向"
        BackColor="#99FFCC" />
    </form>
</body>
public partial class UseMenu : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        //改变菜单显示方向
        if (this.Menu1.Orientation == Orientation.Horizontal)
        {
            this.Menu1.Orientation = Orientation.Vertical;
        }
        else
        {
            this.Menu1.Orientation = Orientation.Horizontal;
        }
    }
}
```

生成并浏览该网页，可以得到如图 8-10 所示的网页，这是 Menu1 竖排显示的效果，同时显示“欢迎页面”菜单的子菜单“欢迎子菜单”。

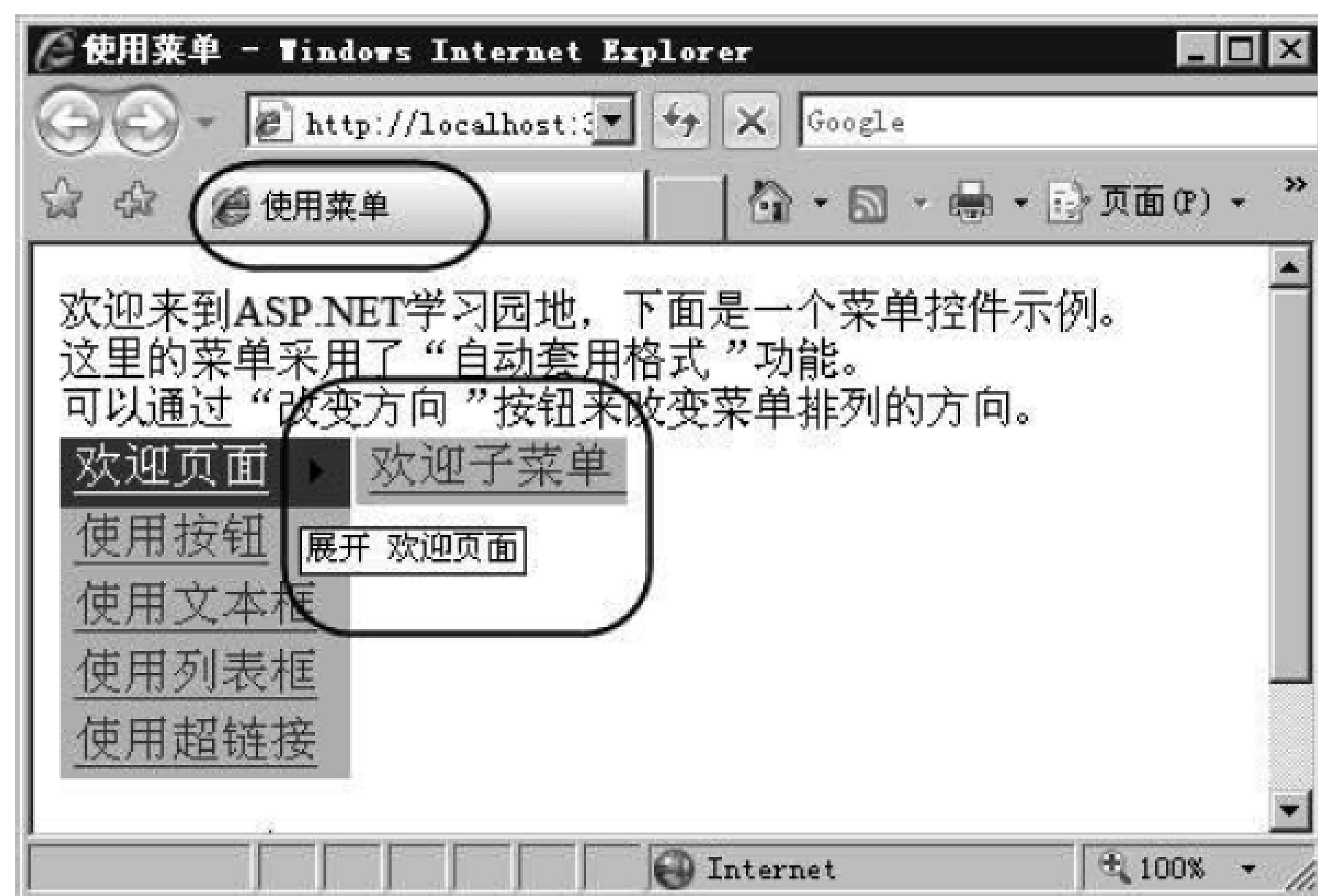


图 8-10　菜单控件运行效果

8.3　网页开发实例——用户注册

前面介绍了常用的 Web 控件，本节以一个简单的用户注册网站作为实例，介绍一个 Web 网站的开发过程，进一步熟悉 ASP.NET 下网站开发的基本步骤和相关细节。

8.3.1　设计用户注册网站

和开发普通的桌面软件一样，在开发一个网站之前，首先需要明确网站的功能需求，即需要做什么？然后再去考虑该如何做？假设在用户注册网站中，需要实现以下功能：

- 用户注册：用户需要指定用户名和密码，并指定个人信息包括姓名、性别、年龄、邮箱、电话和爱好。
- 用户登录：用户根据自己的用户名和密码进行登录。
- 用户查看个人信息：用户在成功登录后可以查看自己的个人信息。

该网站的访问流程如图 8-11 所示，用户第一次进入时显示“欢迎界面”，如果是新用户则通过注册功能进入“注册页面”创建新的用户。如果是老用户首先需要登录，登录成功后可以查看个人信息。

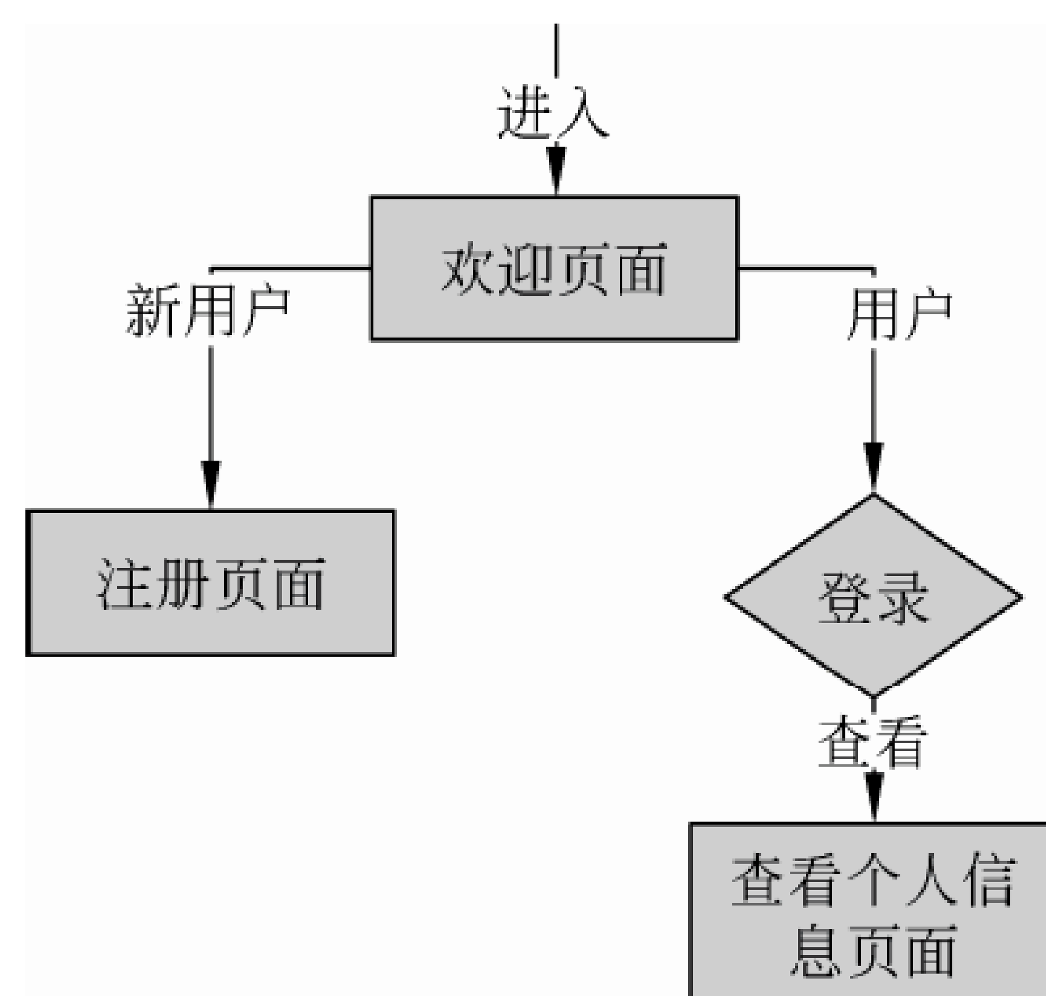


图 8-11　用户注册网站访问流程

基于上面的需求，用户注册网站需要设计一个简单的用户数据库——用户信息，只需要使用 Access 数据库即可。留言信息数据库的数据表信息如表 8-1 所示，包括一个数据库表——用户信息，表“用户信息”包含的字段有用户名、密码、姓名、性别、年龄、电话、邮件。

表 8-1　留言信息数据库表

表　　名	字　段　名	字段类型	字 段 说 明
用户信息 (UserInfo)	用户名（UserID）	文本	长度为 50 的文本，表示登录名，只能是英文字符
	密码（Password）	文本	长度为 10 的文本，表示用户的登录密码
	姓名（UserName）	文本	长度为 10 的文本，表示用户的姓名
	性别（UserXB）	文本	长度为 10 的文本，表示用户的性别
	年龄（UserAge）	整数	表示用户的年龄
	电话（UserTel）	文本	长度为 20 的文本，表示用户的电话号码
	邮件（UserE-mail）	文本	长度为 50 的文本，表示用户的电子邮件

接下来创建一个名为 UserManagement 的 ASP.NET 网站应用程序，具体步骤见 8.1.2 节的描述，大致包括以下 3 个步骤：

（1）创建一个 Web 网站应用程序，命名为 UserManagement。

（2)创建一个具有表 7-1 所示的 Access 数据库文件，命名为 UserManagement.mdb，并将它添加到网站 UserManagement 的 App_Data 目录下（App_Data 目录主要是用来存放网站所需要用到的数据）。

（3）依次添加 3 个页面：欢迎页面、注册页面和查看个人信息页面到网站中，分别命名为 welcom.aspx、register.aspx、viewuser.aspx。最后得到 Web 应用程序 UserManagement 的文件结构如图 8-12 所示。

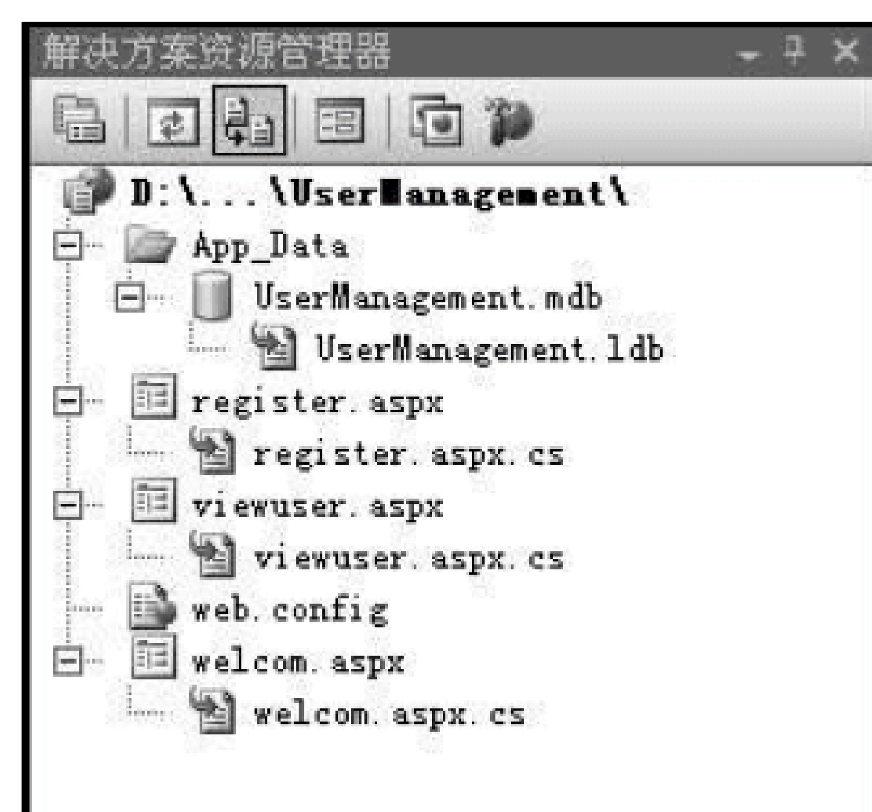


图 8-12　UserManagement 的网站结构

8.3.2　实现欢迎页面 Welcom.aspx

一个独立的网站通常都包含一个“主页/欢迎页面”，该页面通常包括本站的使用帮助信息，重要信息汇总等，对访问者起到一个指导和了解本站概况的作用。在 UserManagement 网站中，欢迎界面只是简单地给出一点提示信息和导航菜单，设计效果如图 8-13 所示。



图 8-13　Welcome.aspx 设计效果

从图 8-13 中可以看出，在 UserManagement 网站中，登录界面主要分为 3 部分：导航菜单、提示信息、登录信息。“管理”菜单下包括“注册用户”子菜单，导航到 Register.aspx，“功能”菜单下包括“查看用户”子菜单，导航到 ViewUser.aspx，以及“编辑用户”子菜单，导航到 EditUser.aspx。

在 Welcom.aspx 中，给出最基本的登录界面，包括“用户名”和“密码”两个 TextBox 控件，分别命名为 tbUserID 和 tbPassword，而且 tbPassword 的 TextMode 属性为 Password，便于隐藏密码。同时还包括一个“登录”按钮 btnLogin。

技巧：.NET 类库也提供了用户登录控件 Login 等，它主要是与.NET 用户认证和角色管理组件同时使用，得到一套系统的用户认证和登录功能。如读者有兴趣，可以自行研究一下该方案。

在实现了前台页面之后，接下来为该页面实现后台逻辑，在 Welcome.aspx 页面中，主要需要实现的是登录功能。添加“登录”按钮 btnLogin 的 Click 事件响应函数 btnLogin_Click()，在该函数中从前面设计好的数据读取用户名和密码，如果通过验证转到 ViewUser.aspx，否则提示信息，并留在当前页面，如示例代码 8-7 所示。

示例代码 8-7

```
using System.Data.OleDb;
public partial class welcom : System.Web.UI.Page
{
    private string _CurUser = "";                        //当前用户名
    //页面加载事件处理函数，从 Sessio 获取数据，并判断登录信息等
    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.Session["UserName"] != null)
        {//从 Session 读取当前登录的用户名
            this._CurUser = (string) this.Session["UserName"];
        }
        if (this._CurUser == "")
        {
            this.lbUser.Text = "游客";                   //没有登录，用户提示为游客
        }
        else
        {
            this.lbUser.Text = this._CurUser;            //登录后，用户设置为用户名
```

```
        }
    }
    private bool IsValidUser(string userid, string password )
    {
        //数据库连接，指定数据库为 App_Data 下面的 UserMangement.mdb
        OleDbConnection con = new OleDbConnection(@"Provider=Microsoft.
        Jet.OLEDB.4.0;" + @"Data Source=" + AppDomain.CurrentDomain.
                                BaseDirectory +@"\App Data\UserManagement.
                                mdb;" +@"Persist
                                Security Info=true");
        string cmdText = string.Format("SELECT UserID FROM UserInfo WHERE
        UserID='{0}' AND Password='{1}'", userid, password);    //SQL 命令
        OleDbCommand cmd = new OleDbCommand(cmdText, con); //数据库命令对象
        try
        {
            con.Open( );                                    //打开数据库连接
            OleDbDataReader dr = cmd.ExecuteReader( ) //执行 SQL 命令读取数据
            if (dr.Read( ))                                 //从数据库读到数据
            { //获取从数据库读取的 UserID 字段
                string ru = (string) dr["UserID"];
                return (ru == userid);
            }
            return false;
        }
        catch (Exception)
        {
            return false;                                  //发生异常返回失败
        }
        finally
        {
            con.Close( );                                  //关闭数据库连接
        }
    }
    protected void btnLogin Click(object sender, EventArgs e)
    {
        string userID = this.tbUserID.Text.Trim( );     //获取用户名
        string pass = this.tbPassword.Text.Trim( );     //获取用户密码
        if (IsValidUser(userID, pass))
        {
            this.lbHint.Text = "";
            //将用户保存到 Session 其他页面可用
            this.Session["UserName"] = userID;
            this.Response.Redirect("./ViewUser.aspx"); //跳到查询用户信息页面
        }
        else
        {                                                 //提示密码错误，重新输入
            this.lbHint.Text = "用户名和密码不正确，请重新输入。";
        }
    }
}
```

注意： 在示例代码 8-7 中，命名空间 System.Data.Oledb 是为了使用类 OleDbConnection 等，这些类用来访问 Access 数据库，关于数据库访问的更多细节，将在后面章节中详细介绍。

从示例代码 8-7 中可以看出，作为登录功能，一定要保证用户完全正确才能登录，所以通过首先对从数据库查询到的用户进行再次判断，保证安全。同时在发生任何异常时都返回 false。另外，Session 是一种用来保持界面状态的常用方法，

8.3.3　实现注册页面 Register.aspx

一个完整的用户注册网站，必然需要用户注册功能，用户注册有一个最基本的功能就是数据的合法性验证，即用户输入的密码是否合法、年龄是否合法等。.NET 类库提供了一些专门用于数据验证的控件，它可以判断数据是否为空，两个数据是否相同，数据是否在某个范围等，同时还可以判断数据是否符合某个正则表达式。

如图 8-14 是 UserMangement 网站的注册页面设计图，标记 1 是一个 LinkButton，通过它可以返回到欢迎页面。标记 2 是一些验证控件，分别对它们左边的数据进行各种检查，如果不合法则提示错误信息，合法就不提示。标记 3 是“注册”按钮，单击它之后，所有的验证控件首先会进行数据验证，如果任何一个验证失败，都不会提交到服务器进行处理。

回到主页 1
请在下面输入用户信息，然后点击提交。
登录信息：
用户名(*)：　请输入合法的用户名
密　码(*)：　2
密码重复：　两次密码要保持一致
详细信息：
姓名：
性别：男
年龄：　请输入0-120之间的数
电话：　请输入合法的电话号码
邮箱：　请输入合法的邮箱
注　册　3

图 8-14　Register.aspx 设计效果

示例代码 8-8 是 Register.aspx 页面的“注册”按钮 Click 事件处理函数的具体实现，它首先从界面上获取用户信息的各个字段数据，然后通过 AddUser()方法将新用户添加到数据库，这里关于数据库操作的更多细节，将在后面章节介绍。用户添加之后，再通过 Session 设置当前登录用户为新用户，并跳转到查询用户信息页面。

示例代码 8-8

```
//将新用户保存到数据库中
private void AddUser(string id, string pd, string name, string xb, int age,
string tel, string email)
{
    //数据库连接，指定数据库为 App_Data 下面的 UserMangement.mdb
    OleDbConnection con = new OleDbConnection(@"Provider=Microsoft.Jet.
    OLEDB.4.0;" +
```

```
                            @"Data Source=" + AppDomain.CurrentDomain.Base
                            Directory +
                            @"\App Data\UserManagement.mdb;" +
                            @"Persist Security Info=true");
    string cmdText = string.Format(
          "INSERT INTO UserInfo(UserID, [Password], UserName, UserXB,
          UserAge, UserTel, UserEmail)" +"VALUES('{0}', '{1}', '{2}', '{3}',
          {4}, '{5}', '{6}')",
          id, pd, name, xb, age, tel, email);                 //SQL 命令
    OleDbCommand cmd = new OleDbCommand(cmdText, con);        //数据库命令对象
    try
    {
       con.Open( );                                           //打开数据库连接
       cmd.ExecuteNonQuery( );                                //执行命令
    }
    catch (Exception)
    {}
    finally
    {
       con.Close( );                                          //关闭数据库连接
    }
}
//注册按钮 Click 事件处理函数，从界面获取数据，并保存到数据库
protected void btnReg Click(object sender, EventArgs e)
{
    string password1 = this.tbPass1.Text;                 //第一个密码
    string password2 = this.tbPass2.Text;                 //密码确认
    if (password1 != password2)                           //两次密码不同，返回
       return;
    string userName = this.tbUserID.Text.Trim( );         //获取用户 ID
    string name = this.tbName.Text.Trim( );               //获取姓名
    string xb = this.cmdbXB.Text;                         //获取性别
    int age = int.Parse(this.tbAge.Text);                 //获取年龄
    string tel = this.tbTel.Text.Trim();                  //获取电话
    string email = this.tbEmail.Text.Trim( );             //获取邮箱
    this.AddUser(userName, password1, name, xb, age, tel, email);
                                                          //保存用户到数据库
    this.Session["UserName"] = userName;                  //设置 Session，表示
                                                            新用户登录
    this.Response.Redirect("./ViewUser.aspx");            //转到查看用户页面
}
```

技巧： 由于 Register.aspx 在界面上对所有的用户注册信息用验证控件进行了验证，所以在后台逻辑代码上不需要进一步验证合法性，但是密码属于比较重要的数据，所以重新进行一次验证。

8.3.4　实现查看用户页面 ViewUser.aspx

如图 8-15 是 UserMangement 网站的查看用户信息页面设计图，标记 1 是一个 LinkButton，通过它可以返回到欢迎页面。标记 2 是一个 Label 控件 lbUser，它用来提示当前登录的用户，如果没有用户登录则显示“游客”。标记 3 是一批只读的文本框控件，分别表示当前登录用户的各个信息。



图 8-15　ViewUser.aspx 设计效果

注意： 由于是查看用户信息页面，所以为了安全性，用户的密码不能被查看，所以不显示，就算显示到页面上，也应该实现为 Password 模式。

示例代码 8-9 为 ViewUser.aspx 页面的后台实现代码，它在加载的时候通过 Page_Load()方法从 Session 中获取当前登录用户，如果没有用户登录，将 lbUser 设置为“游客”。如果有用户登录，将 lbUser 设置为登录用户名，然后通过 ShowUser()方法从数据库获取登录用户信息，并显示到页面上。

示例代码 8-9

```
public partial class viewuser : System.Web.UI.Page
{
    private string  _CurUser = "";
    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.Session["UserName"] != null)    //获取当前登录的用户
        {
            this._CurUser = (string) this.Session["UserName"];
        }
        if (this._CurUser == "")
        {
            this.lbUser.Text = "游客";               //如果没有用户登录，设置为游客
        }
        else
        {
            this.lbUser.Text = this._CurUser;    //提示当前登录用户
            this.ShowUser( );                    //显示当前登录用户的信息
        }
    }
    //从数据库读取当前登录用户的信息，并显示到页面上
    private void ShowUser( )
    {
        //数据库连接，指定数据库为 App_Data 下面的 UserMangement.mdb
        OleDbConnection con = new OleDbConnection(@"Provider=Microsoft.Jet.
        OLEDB.4.0;" +@"Data Source=" + AppDomain.CurrentDomain.Bas
                                  eDirectory +@"\App_Data\UserManagement.mdb;" +
                                  @"Persist Security Info=true");
        string cmdText = string.Format(
                             "SELECT * FROM UserInfo WHERE UserID = '{0}'",
                             this._CurUser);                          //SQL 命令
        OleDbCommand cmd = new OleDbCommand(cmdText, con);  //数据库命令对象
        try
```

```
        {
            con.Open( );                                        //打开数据库连接
            OleDbDataReader dr = cmd.ExecuteReader( );          //读取数据
            if (dr.Read( ))                                     //有数据，则读取并显
                                                                 示到界面
            {
                this.tbUserID.Text = (string)dr["UserID"];
                this.tbName.Text = (string) dr["UserName"];
                this.tbXb.Text = (string) dr["UserXB"];
                this.tbAge.Text = ((int) dr["UserAge"]).ToString( );
                this.tbTel.Text = (string) dr["UserTel"];
                this.tbEmail.Text = (string) dr["UserEmail"];
            }
        }
        catch (Exception)
        { }
        finally
        {
            con.Close( );                                       //关闭数据库连接
        }
    }
}
```

8.3.5　发布用户注册网站

到此为止，已经创建好了一个简单的用户注册网站，并且在调试环境下可以正常运行了，接下来要做的就是如何将网站展示给访问者，这就需要发布网站。在 Visual Studio 2010 中，要发布网站只需要简单的 6 个步骤就可以完成：

（1）确保 Web 应用程序没有错误，即至少可以调试正常运行。确保操作系统上已经安装了 IIS 服务器组件。

（2）在“解决方案管理器”中选中 Web 项目右击，本例中选中 UserManagement，在弹出的快捷菜单中选择“发布网站”选项，弹出“发布网站”对话框。

（3）在左侧选择目标位置为“本地 IIS 服务”，此时右侧默认在“本地 Web 服务器”，这里是安装在机器上的所有 Web 服务器，不同的操作系统此处会略有不同。最后选择“默认网站”，单击“打开”按钮，回到“发布网站”对话框。



图 8-16　选择发布位置对话框

（4）在“发布网站”对话框中单击“确定”按钮，开始发布。在“输出”窗口中将看到发布过程，大致会得到如下的过程提示。

```
------ 已启动生成: 项目: D:\...\UserManagement\, 配置: Debug Any CPU ------
正在预编译网站

正在生成目录"/UserManagement/"。
预编译完成
------ 发布已启动: 项目: D:\...\UserManagement\, 配置: Debug Any CPU ------
正在连接到站点 http://localhost...
正在删除现有文件...
正在发布目录/...
正在发布目录 App_Data...
正在发布目录 bin...
========== 生成: 成功或最新 1 个，失败 0 个，跳过 0 个==========
========== 发布: 成功 1 个，失败 0 个，跳过 0 个==========
```

（5）通过上面 4 个步骤完成网站的发布之后，可以在“计算机管理”的 IIS 服务器目录下看到前面开发的网页，如图 8-17 所示，比如 welcom.aspx、register.aspx 等。另外，App_Data 文件夹还包括数据库文件 UserManagement.mdb。Bin 文件夹下是自动生成的 DLL 文件，它们与 aspx 脚本文件一起共同组成了该网站的所有程序，此时已经可以通过 http://127.0.0.1/welcom.aspx 访问欢迎页面，但是网址 http://127.0.0.1 还不能正常访问。



图 8-17　IIS 默认网站视图

（6）为了让 http://127.0.0.1 能正常访问，还需要设置 IIS 服务器的默认文档属性。在图 8-17 的对话框中选中“默认网站”并右击，在弹出的快捷菜单中选择“属性”选项。在弹出的“默认网站属性”对话框中选择“文档”选项卡，并选中“启用默认文档”复选框。通过“添加”按钮，将留言本网站的默认网页 welcome.aspx 添加到列表中，如图 8-18 所示。此时可以通过 http://127.0.0.1 访问留言本网站。



图 8-18　设置默认启动页

注意：ASP.NET 网站第一次运行时对 ASPX 页面代码进行解析和编译，所以第一次访问会比较慢，但之后不需要编译所以运行速度很快。当页面、二进制代码、配置等信息发生了更改时，会再次编译。

8.4　小　　结

本章简单介绍了 ASP.NET 开发 Web 应用程序的基本概念，并介绍了 ASP.NET 与 C#集成开发的方法。介绍了 ASP.NET 开发中常用的 Web 服务器控件，及它们的使用示例。最后通过一个简单的用户注册网站介绍了网站的设计、开发和发布的全过程。通过本章的学习，读者应该掌握以下知识点：

- ASP.NET 是什么？它具有哪些优势？
- 如何通过 Visual Studio 进行 ASP.NET 网页开发？
- 网站开发中常用的 Web 服务器控件有哪些，如何使用？
- 网站设计的过程是怎样的？
- 如何发布网站？

第 3 篇　SQL Server 2008 基础

第 9 章　SQL Server 2008 入门

随着计算机软件的应用越来越广泛，软件需要处理的数据也越来越复杂，目前中小应用软件中数据存储主要还是依靠关系型数据库来完成。SQL Server 是微软推出的关系型数据库管理软件，也是当今软件开发中使用最为流行的几种关系型数据库之一，经过多年的发展，不断完善和改进，目前已经进入 SQL Server 2008 时代。本章将介绍如何通过 SQL Server 2008 管理数据库。

9.1　什么是 SQL Server

SQL Server 作为微软推出的数据库管理软件，已经发展到 SQL Server 2008，该版本在性能、稳定性、友好性等多个方面都远胜于早期版本。本节将简单介绍 SQL Server 2008 的主要功能。

9.1.1　了解关系数据库

随着计算机软件在实际社会中的广泛应用，软件所要处理的数据越来越多样化，数据间的关系也越来越复杂，数据库的管理和处理在软件开发过程中变得越来越困难和重要。那么什么是“数据库”呢？数据库是存储在一起的相关数据的集合，这些数据是结构化的、无有害的或不必要的冗余，并为多种应用服务，同时还包括这些数据之间的关系。数据的存储独立于使用它的程序，对数据库插入新数据、修改和检索原有数据均能按一种公用的和可控制的方式进行。

数据库通常分为 3 种：层次数据库、网络数据库和关系数据库，其中关系数据库是目前应用最广泛的一种，它将所有数据的关系都看成是二元关系。当今流行的关系数据库中主要有两种：桌面数据库和客户/服务器数据库。一般而言，桌面数据库用于小型的单机应用程序，它不需要网络和服务器，实现起来比较方便，但它只提供数据的存取功能，如 Access、FoxPro 等。客户/服务器数据库主要适用于大型的、多用户的数据库管理系统，应用程序的一部分驻留在客户机上，用于向用户显示信息及实现与用户的交互。另一部分驻留在服务器中，主要用来实现对数据库的操作和对数据的计算处理。例如 SQL Server、Oracle 等。

表（Table）和字段（Field）是关系数据库的两个核心概念，在关系数据库中，表采用二维表格来存储数据，是一种按行与列排列的具有相关信息的逻辑组。如表 9-1 所示，该表用二维关系表示并存储用户的基本信息。一个数据库可以包含任意多个数据表，表之间通常存在一定的关系。

表 9-1　关系数据库Table示例

用　户　名	姓　　名	邮　　箱
Lisi	李四	Lisi@test.com
Zhangsan	张三	Zhangsan@test.com
Wangwu	王五	Wangwu@test.com

数据表中的每一列称为一个字段（Field），如表 9-1 中的用户名、姓名和邮箱。表中的每个字段描述了它所含有数据的意义、类型、大小等，通常字段可以包含各种字符、数字甚至图形，根据不同的数据库管理软件会有所不同。多个字段组合到一起定义了表的结构。关系数据库通过主键（Primary Key）确保表中记录的唯一性，每个表都应该有一个主键，它可以是表中的一个字段或多个字段，常用作一个表的索引字段。数据库表中的一行数据被称为一条数据库记录（Record），如表 9-1 中的“张三”这一行数据就是表示“张三”这个用户的一条记录。每条记录的关键字都是不同的，因而可以唯一地标识一个记录。

索引（Index）是表中单列或多列数据的排序列表，每个索引指向其相关数据表的某一行，通过索引可以在数据库中快速查找数据。索引提供了一个指向存储在表中特定列的数据指针，然后根据所指定的排序顺序排列这些指针。

关系数据库除了存储数据外，还有一个核心功能就是表示数据之间的关系。一个数据库往往都包含多个表，不同类别的数据存放在不同的表中。关系（Relationship）把各个表联接起来，将来自不同表的数据组合在一起。表与表之间的关系是通过各个表中的某一个主键建立起来的，建立表关系所用的关键字段应具有相同的数据类型和大小等。

SQL Server 2008 作为最常用的关系数据库之一，具有操作方便、易于维护、安全快捷等特点。在 9.1.2 节将详细介绍 SQL Server 2008 的特点。

9.1.2　了解 SQL Server 2008

SQL Server 2008 是一个全面、集成、端到端的数据解决方案，它为用户提供了一个安全、可靠和高效的平台用于企业数据管理和商业智能应用。SQL Server 2008 为数据处理和开发者带来强大而熟悉的管理软件，同时降低了数据系统的多平台上创建、部署、管理、使用和分析的复杂度。通过全面的功能集和现有系统的集成性及对日常任务的自动化管理能力，SQL Server 2008 为不同规模的企业提供了一个完整的数据解决方案。

图 9-1 所示为 SQL Server 2008 数据平台的组成架构，SQL Server 2008 数据平台包括以下工具。

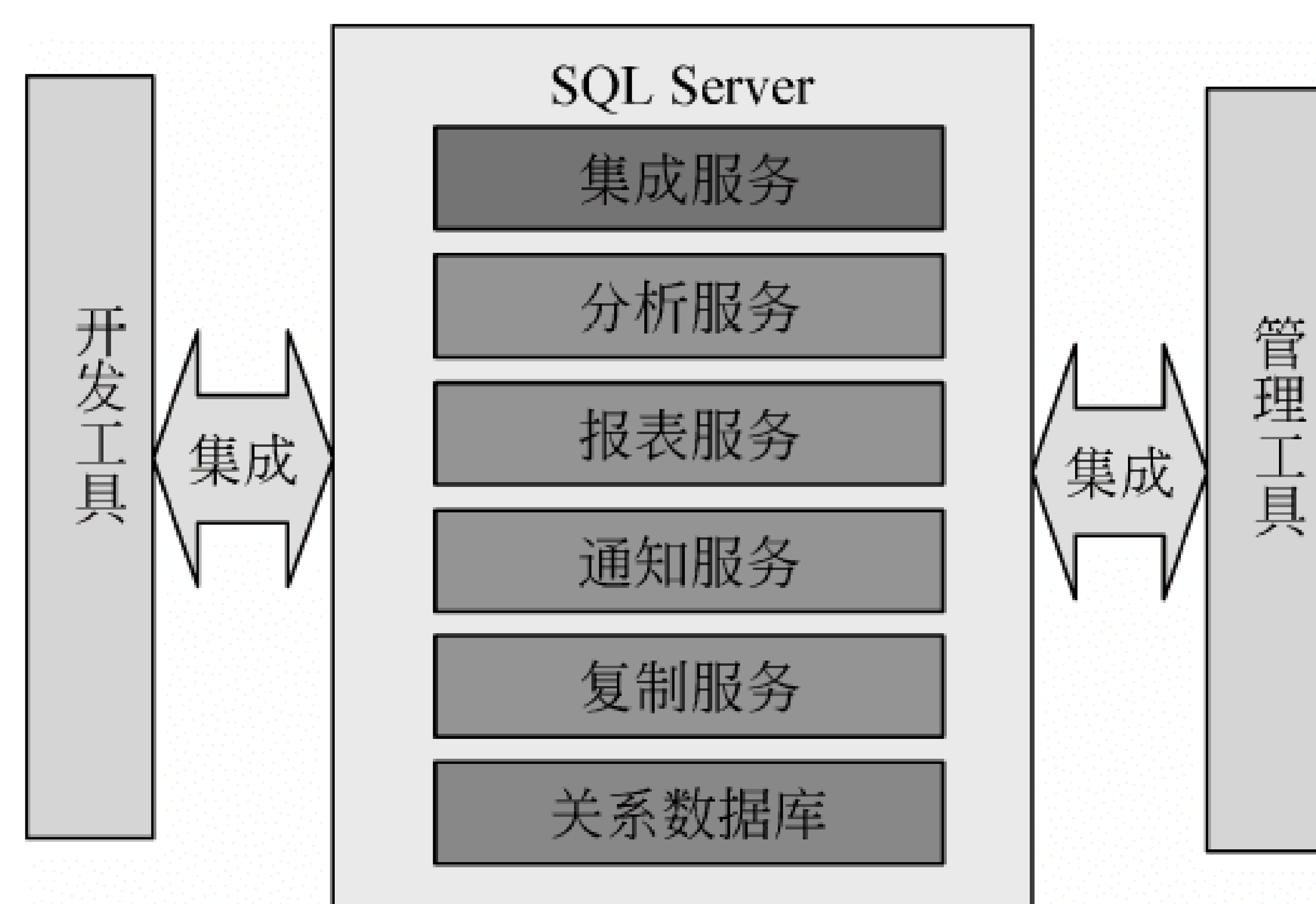


图 9-1　SQL Server 2008 数据平台

- 关系型数据库（Relational DataBase）：安全、可靠、可伸缩、高可用的关系型数据库引擎，提升了性能且支持结构化和非结构化（XML）数据。
- 复制服务（Replication Services）：数据复制可用于数据分发、处理移动数

据应用、系统高可用、企业报表解决方案的后备数据可伸缩存储、与异构系统的集成等，包括已有的 Oracle、Access 等数据库。

- 通知服务（Notification Services）：用于开发、部署可伸缩应用程序的先进的通知服务，能够向不同的连接和移动设备发布个性化、及时的信息更新。
- 集成服务（Integration Services）：可以提供数据仓库和企业范围内数据集成的抽取、转换和装载等功能。
- 分析服务（Analysis Services）：联机分析处理功能可用于多维存储的大量、复杂的数据集的快速高级分析。
- 报表服务（Reporting Services）：全面的报表解决方案，可创建、管理和发布传统的、可打印的报表和交互的、基于 Web 的报表。
- 管理工具（Management tools）：SQL Server 2008 包含的集成管理工具可用于高级数据库管理，它也和其他微软工具紧密集成在一起。标准数据访问协议大大减少了 SQL Server 和现有系统间数据集成所花的时间。此外，构建于 SQL Server 内的内嵌 Web service 确保和其他应用及平台的互操作能力。
- 开发工具（Development Tools）：SQL Server 2008 为数据库引擎、数据抽取、转换和装载（ETL）、数据挖掘、OLAP 和报表提供了和 Microsoft Visual Studio 相集成的开发工具，以实现端到端的应用程序开发能力。SQL Server 中每个主要的子系统都有自己的对象模型和 API，能够以任何方式将数据系统扩展到不同的商业环境中。

SQL Server 2008 通过不同的功能组件，为用户提供完善的数据管理和分析支持，它所包含的内容非常多，本书只是介绍它最基本的功能——数据库管理，同时介绍它与 Visual Studio 2005 的集成。关于 SQL Server 2008 其他组件及更加详细的信息，读者可以参见联机帮助或是相关网站和书籍。

说明： SQL Server 的安装比较简单，这里不再给出详细的安装步骤。

9.2　SQL Server 管理工具

数据库管理是 SQL Server 2008 最基本的功能，也是中小应用软件和日常开发工作中最常用的功能，本节将介绍如何通过 SQL Server 2008 的管理工具（Management Tool）管理数据库。

9.2.1　SQL Server Management Studio 管理器

Microsoft SQL Server Management Studio 是 Microsoft SQL Server 2008 提供的一种新集成环境，用于访问、配置、控制、管理和开发 SQL Server 的所有组件。SQL Server Management Studio 将一组多样化的图形工具与多种功能齐全的脚本编辑器组合在一起，可为各种技术级别的开发人员和管理员提供对 SQL Server 的访问。

SQL Server Management Studio 将以前企业管理器和查询分析器功能整合到单一环境

中。此外，开发人员可以获得熟悉的体验，而数据库管理员可获得功能齐全的单一实用工具，其中包含易于使用的图形工具和丰富的脚本撰写功能。Microsoft SQL Server Management Studio 包括以下常用功能：

- ❑ 支持 SQL Server 2008 和 SQL Server 2005 的大部分数据库管理任务。
- ❑ 用于 SQL Server Database Engine 管理和创作的单一集成环境。
- ❑ 提供用于管理 SQL Server 数据库引擎、分析服务、报表服务、分析服务，以及 SQL Server Mobile 中对象的新管理对话框，使用这些对话框可以立即执行操作，还可以将操作发送到代码编辑器或将其编写为脚本供以后执行。
- ❑ 非模式及大小可调的对话框，允许在打开某一对话框的情况下访问多个工具。常用的计划对话框可以在以后执行管理对话框的操作。
- ❑ 在 Management Studio 环境之间导出或导入 SQL Server Management Studio 服务器注册。
- ❑ 集成的 Web 浏览器可以快速浏览 MSDN 或从网上社区取得联机帮助。
- ❑ SQL Server Management Studio 教程可以帮助用户充分利用许多新功能，并可以快速提高效率。

SQL Server Management Studio 提供的功能全面，也有一些并不常用，本章将主要介绍它的数据库管理功能，包括创建数据库、数据表、查询、存储过程等。

在安装了 SQL Server 2008 之后，可以在“开始”|“程序”|“Microsoft SQL Server 2008”菜单下找到 SQL Server Management Studio 的启动菜单，通过该菜单启动。首先需要选择将要连接的数据库服务器，如图 9-2 所示，这里可以连接数据库引擎、分析服务、报表服务、移动 SQL Server 和集成服务 5 种服务器类型，本章所有实例都选择“数据库引擎”。



图 9-2　连接到数据库服务器

图 9-2 中，在“服务器名称”下拉列表框中选择或输入数据库服务器名称。在“身份验证”一栏选择身份验证类型，包括“Windows 身份验证”和“SQL Server 身份验证”两种，前者使用当前登录的操作系统用户名和密码登录到数据库服务器，后者需要指定登录的用户名和密码。本例中采用“Windows 身份验证”。最后，单击“连接”按钮连接到指定的数据库服务器。

注意：Windows 身份验证常用于安装在本机的数据库服务器，同时 SQL Server 在安装时允许 Windows 身份验证。对于其他服务器上的数据库，通常需要采用 SQL Server 身份验证。

数据库服务器成功连接后，得到如图 8-8 所示的管理界面，可以看出它的整体框架和 Visual Studio 2010 非常相似，使用也非常方便。其中，“菜单栏”和“工具栏”都会根据当前操作的对象进行自动更新，也可自定义指定“工具栏”。

“对象资源管理器”是 Management Studio 的核心视图，它用树状结构列出了可以管理（或操作）的对象，“数据库”子结点列出了当前服务器所有的数据库，可以选择任何一个进行编辑。安全性、服务器对象、复制、管理等都是针对数据库服务器本身进行的管理功能。

“摘要”视图是一个实时更新的视图，它会自动给出当前在“对象资源管理器”视图中选中对象的汇总信息。在图 9-3 中，由于选中了数据库 AdventureWorks 的“表”字结点，所以“摘要”视图列出了数据库 AdventureWorks 中所有的表及相关信息。

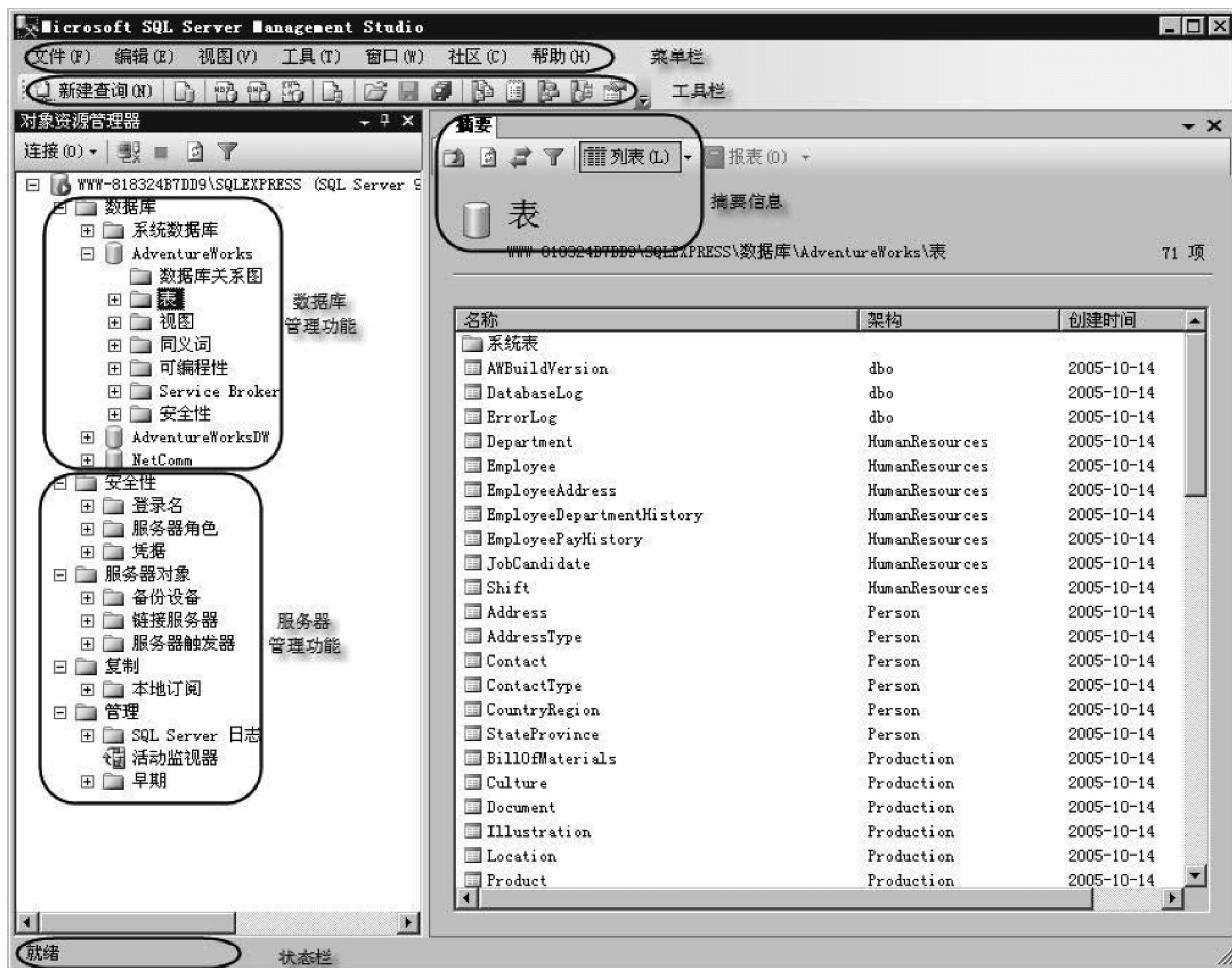


图 9-3　Management Studio 主要界面

9.2.2　创建数据库——UserLog

在通过 9.2.1 节介绍的方法连接到数据库之后，接下来创建一个数据库——UserLog，

本例中 UserLog 是一个保存用户日志的数据库，用于记录用户信息和用户的留言信息。在 SQL Server Management Studio 中创建数据库非常简单，只需要以下几个步骤即可：

（1）按照 9.2.1 节介绍的方法连接到需要的数据库服务器。

（2）在“对象资源管理器”视图中选中“数据库”子结点并右击，弹出数据库相关操作菜单，如图 9-4 所示，包括如下 5 个功能。

- 新建数据库：创建一个新的数据库，本节将使用该命令。
- 附加：将一个已有的数据库文件（*.mdf）附加到当前数据库中。
- 还原数据库：对某个已有数据库的数据进行还原。
- 还原文件和文件组：从已经损坏的数据库文件和文件组中还原数据。
- 刷新：刷新“数据库”子结点下的所有子结点，即刷新数据库的信息。

（3）在图 9-4 中选择“新建数据库”菜单，弹出如图 9-5 所示的“新建数据库”对话框，包含“常规”、“选项”、“文件组”3 个分项需要设置。在“常规”选项卡的“数据库名称”文本框中输入新的数据库名称，这里为 UserLog。如果需要用户权限保护，还需要选择数据库的拥有者，这里选择默认值。

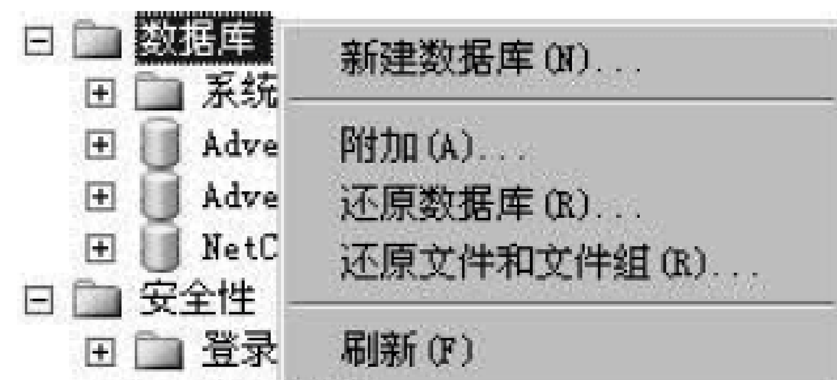


图 9-4 数据库操作菜单

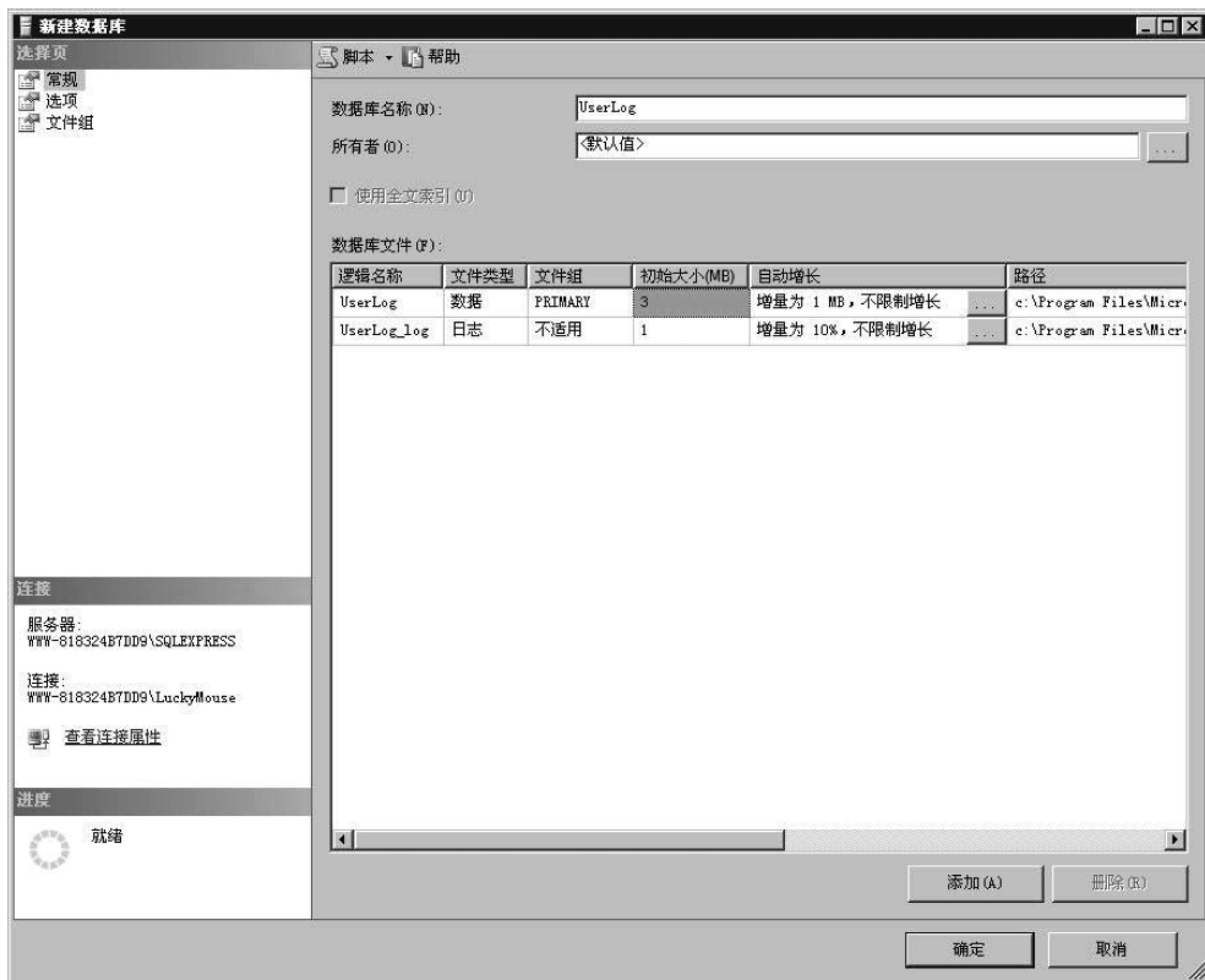


图 9-5 新建数据库对话框

（4）单击图 9-5 中的“确定”按钮完成新数据库——UserLog 的创建。回到“对象资源管理器”可以看到，新建的数据库——UserLog 被列在“数据库”子结点下面。

展开数据库 UserLog 的“表”子结点，可以看到里面并不包含任何数据表（Table），也不包括任何视图、关系图等，后面几节将介绍如何创建数据库的这些结构。

9.2.3　创建数据表——Users

在添加了数据库之后，就需要为数据库添加具体的表（Table），表是关系型数据库中实际保存数据的基本单元。在数据库 UserLog 中，需要添加两个数据表，如表 9-2 所示。

表 9-2　UserLog数据库表

表名	字　段　名	字 段 类 型	字 段 说 明
Users	LoginID	文本	表示用户登录用户名
	Password	文本	表示用户的密码
	Name	文本	表示用户的姓名
	Age	整数	表示用户的年龄
	XingBie	文本	表示用户的性别
	Mobile	文本	表示用户的手机号码
	Email	文本	表示用户的电子邮箱地址
Logs	ID	整数	唯一表示一个用户的日志信息
	UserID	文本	用户 ID，表示留下当前日志的用户
	LogContent	文本	表示日志的具体内容
	LogTime	日期时间	表示用户留下该日志的时间

在 SQL Server 2008 Management Studio 中，可以通过如 4 步为数据库 UserLog 添加一个名为 Users 的数据表。

（1）展开要添加表的数据库，右击它的“表”子结点，在弹出的快捷菜单中选择“新建表”选项，打开新建表视图，如图 9-6 所示。标记 1 为创建数据表专用工具栏，可以指定表的主键、关系图等。标记 2 和 3 给出了当前编辑的数据表——Users。



图 9-6　设计数据表视图

（2）在图 9-6 中，标记 4、5、6、7 构成了数据表设计的主要视图，可以在“列名”中直接输入或修改数据表的字段名，在“数据类型”列中直接选择字段的数据类型。如果“允许空”被选中，表示该字段可以为空，否则不能为空。在这里，添加了 LoginID 列到表 Users 中，LoginID 数据类型为 int，不允许为空等。表 Users 中 LoginID 列为主键，它的左边有一把钥匙（Key）作为标记。

（3）在图 9-6 中，标记 8 所示的属性窗体给出了当前正在编辑的数据表的属性，包括名称、数据库名称、服务器名等信息。这里名称为 Users，数据库名称为 UserLog。

（4）添加完成数据库中所有的列之后，可以在“对象资源管理器”中找到新建的表结点（Users），展开可以看到它包含 6 个子结点：列、键、约束、触发器、索引、统计信息。双击这些结点中的元素可以在设计窗口中查看和设置它们的属性，另外通过右键菜单，还可以管理列、索引等表信息。

这里重复利用前面的 4 个步骤，为 Users 数据库添加表 9-2 所示的数据表 Logs。当数据表需要更改时，同样可以通过这种方式修改数据表的信息，只需要省去新建步骤即可。

9.2.4　创建关系——ULRleation

关系是关系型数据库的一个重要元素，在添加了数据库 UserLog 的所有数据表 Users 和 Logs 之后，就可以为它们添加关系。在 SQL Server Management Studio 中，关系用关系图来表示，一个关系图可以包含两个或多个数据表的关系。可以通过以下 5 个步骤为数据库添加关系图。

（1）展开要添加关系的数据库 UserLog，右击它的“数据库关系图”子结点，在弹出的快捷菜单中选择“新建数据库关系图”选项，打开“添加表”对话框，如图 9-7 所示。

（2）图 9-7 所示的“添加表”对话框列出了所有可以添加到关系图的数据表，这里选择 Logs 和 Users，单击“添加”按钮，进入关系图设计界面，如图 9-8 所示。



图 9-7　“添加表”对话框

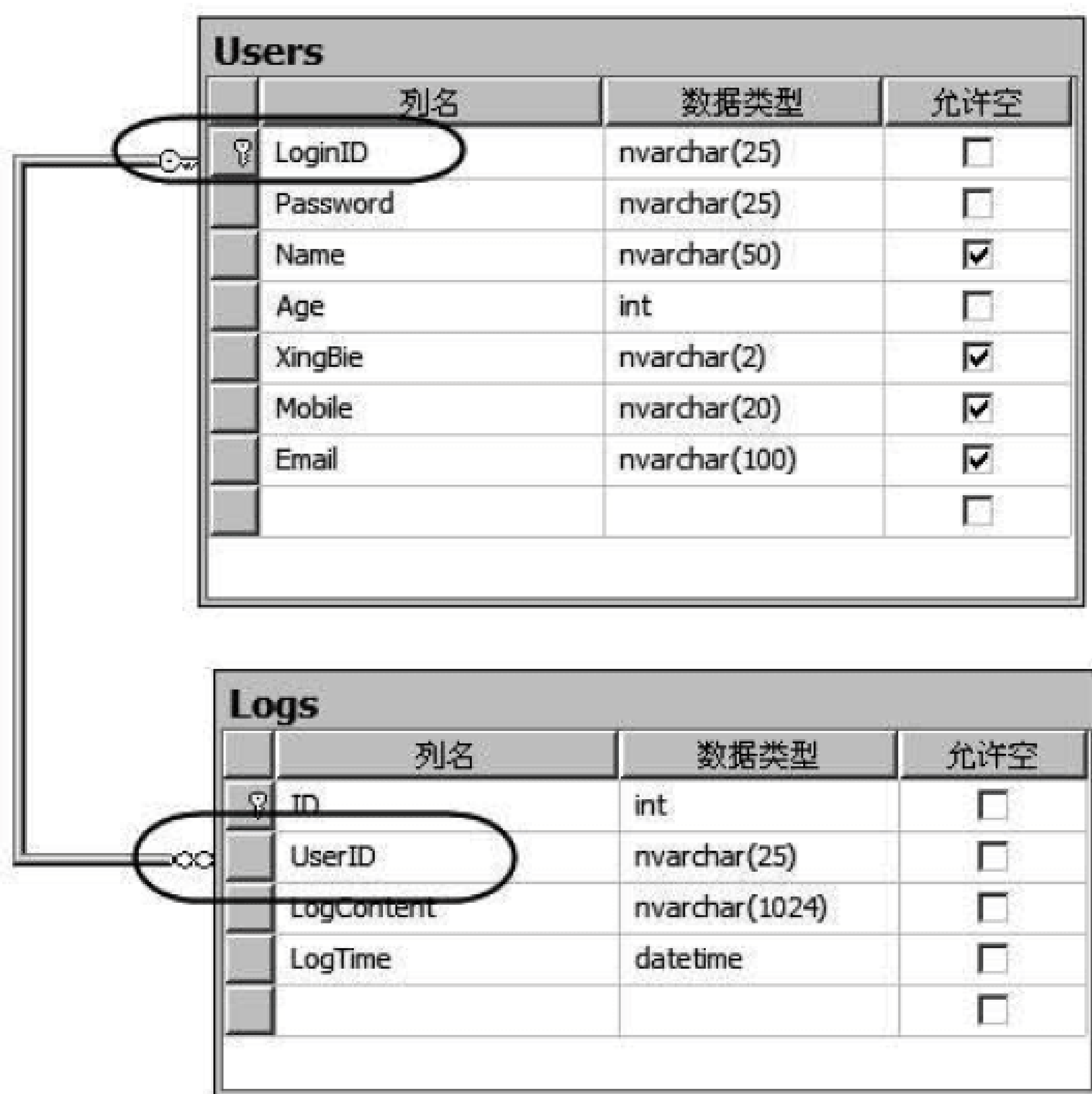


图 9-8　数据库关系设计视图

（3）在图 9-8 所示的关系设计图中可以通过鼠标拖放来排列数据表，可以通过右键菜单“表视图”选择数据表的显示格式：标准、列名、键、仅表名等。

（4）在图 9-8 中，通过鼠标将表 Users 的 LoginID 字段，拖曳到表 Logs 的 UserID 字段，释放鼠标，会弹出关系设计对话框，可以选择相互关系的数据表和字段。设计了数据库关系之后，会在相互联系的数据表之间产生一条线连接起来，如图 9-8 所示，连接线有钥匙的一端表示关系的主键，另外一端表示关系的外键。

（5）设计好数据库关系之后，通过“保存”菜单或工具栏按钮可以命名并保存该数据库关系图。

注意：在数据库关系中，作为关系的两个字段必须具有相同的数据类型（包括类型和长度），如果类型不同，则不能建立关系。所以本例中，Users 的 LoginID 字段和 Logs 的 UserID 字段必须具有相同的数据类型，这里为 nvarchar(25)。

9.2.5 创建视图——ULView

视图（View）是关系型数据库的一个常用概念，它是一种将相互关联的数据组合从数据库中临时提取出来作为一个数据表处理的方式。在 SQL Server Management Studio 中，可以通过以下 3 个步骤为数据库添加一个视图。

（1）展开要添加关系的数据库 UserLog，右击它的“视图”子结点，在弹出的快捷菜单中选择“新建视图”选项，打开“添加表”对话框，如图 9-7 所示。

（2）在图 9-7 中选择要建立查询的数据表，这里选择 Users 和 Logs，然后单击“确定”按钮，进入视图设计界面如图 9-9 所示。

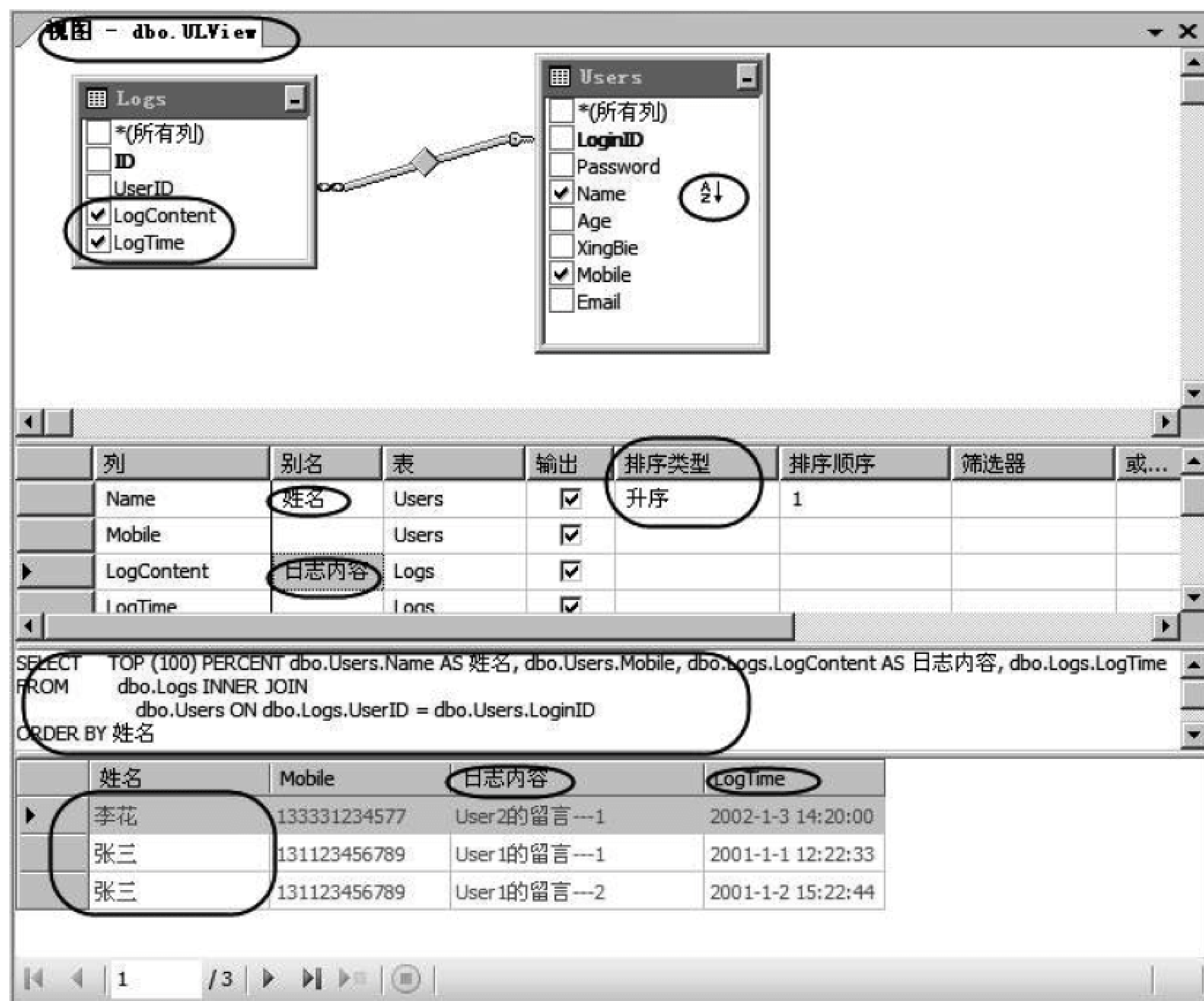


图 9-9　视图（View）设计界面

（3）在图 9-9 所示的视图中设计视图要查询的字段、关系信息等，比如排序、分组等信息。然后通过“保存”菜单或工具栏按钮可以命名并保存该视图。

图 9-9 的设计视图主要分成 4 个部分，从上到下，第 1 部分是可视化的表视图，每个字段的左边选择框可以选择是否显示该字段，每个字段的右边会有标签表示该字段的排序、分组等信息。第 2 部分是以表格的格式给出的字段，其中“别名”列表示显示字段的别名，等价于 SQL 语句的 AS 子句，“排序类型”列表示字段的排序类型，包括升序、降序、不排序 3 个选择。第 3 个部分是该查询的 SQL 语句，它是 SQL Server 最终要执行的查询代码。这 3 个部分是完全同步的，在任何一个部分修改视图都会同步刷新到其他两个部分。

图 9-9 的最下面部分是查询的生成结果，默认情况下没有数据，只有通过“执行”工具栏或菜单命令之后才会产生该视图所产生的数据集。这里的列名字段的列名或别名，如 Users 表的 Name 字段就被显示为别名“姓名”。

9.3　Visual Studio 2010 管理数据库

Visual Studio 2010 与 SQL Server 2008 数据库可以无缝集成，所以可通过 Visual Studio 2010 来管理数据库，包括 Access、SQL Server 2000/2005、Oracle 等。本节将介绍如何通过 Visual Studio 2010 管理数据库。

9.3.1　用 Visual Studio 2010 创建数据库

可以通过 Visual Studio 2010 创建 SQL Server 数据库，并对它进行管理。通过 Visual Studio 2010 创建一个新的 SQL Server 数据库需要以下 4 个步骤。

（1）打开 Visual Studio 2010 开发环境，通过菜单“视图”|“服务器资源管理器”打开“服务器资源管理器”窗口。

（2）在“服务器资源管理器”中右击“数据连接”结点，在弹出的快捷菜单中选择“创建新的 SQL Server 数据库”选项，弹出“创建新的 SQL Server 数据库”对话框，如图 9-10 所示。



图 9-10　创建新 SQL Server 数据库

（3）在“服务器名”下拉列表框中选择目标 SQL Server 服务器名称，也可以通过“刷新”按钮自动获取当前网络中可用的 SQL Server 服务器，包括 SQL Server 2000 和 SQL Server 2008。在“数据库名称”文本框中输入新数据库的名称，如“UserLog2”。

（4）单击“确定”按钮创建新的 SQL Server 数据库。此时可以在“数据连接”结点下看到新建的数据库 UserLog2。当然，通过 SQL Server Management Studio 也可以看到新建的数据库 UserLog2。

注意：目前，Visual Studio 2010 只支持创建 SQL Server 数据库，Access 数据库可以先通过 Microsoft Access 软件创建数据库。

9.3.2　用 Visual Studio 2010 连接到数据库

通过 Visual Studio 2010 还可以对数据库进行管理，设计数据表、修改数据记录、设计视图、设计索引等，这些操作都与 SQL Server Management Studio 中的操作完全一样。但是在进行数据库管理之前，需要将数据库连接到 Visual Studio 2010，这需要通过以下 5 个步骤来完成。

（1）打开 Visual Studio 2010 开发环境，通过菜单“视图”|“服务器资源管理器”打开“服务器资源管理器”窗口。

（2）在“服务器资源管理器”中右击“数据连接”结点，在弹出的菜单中选择“添加连接”选项，弹出“添加连接”对话框，如图 9-11 所示。

（3）单击“更改”按钮从弹出的“更改数据源”对话框（如图 9-12 所示）中选择需要连接的数据源类型，这里选择 Microsoft SQL Server。从图 9-12 中可以看出，Visual Studio 2010 支持 Access、ODBC、SQL Server 服务器、SQL Server 文件、Oracle 等多种类型数据库的管理。

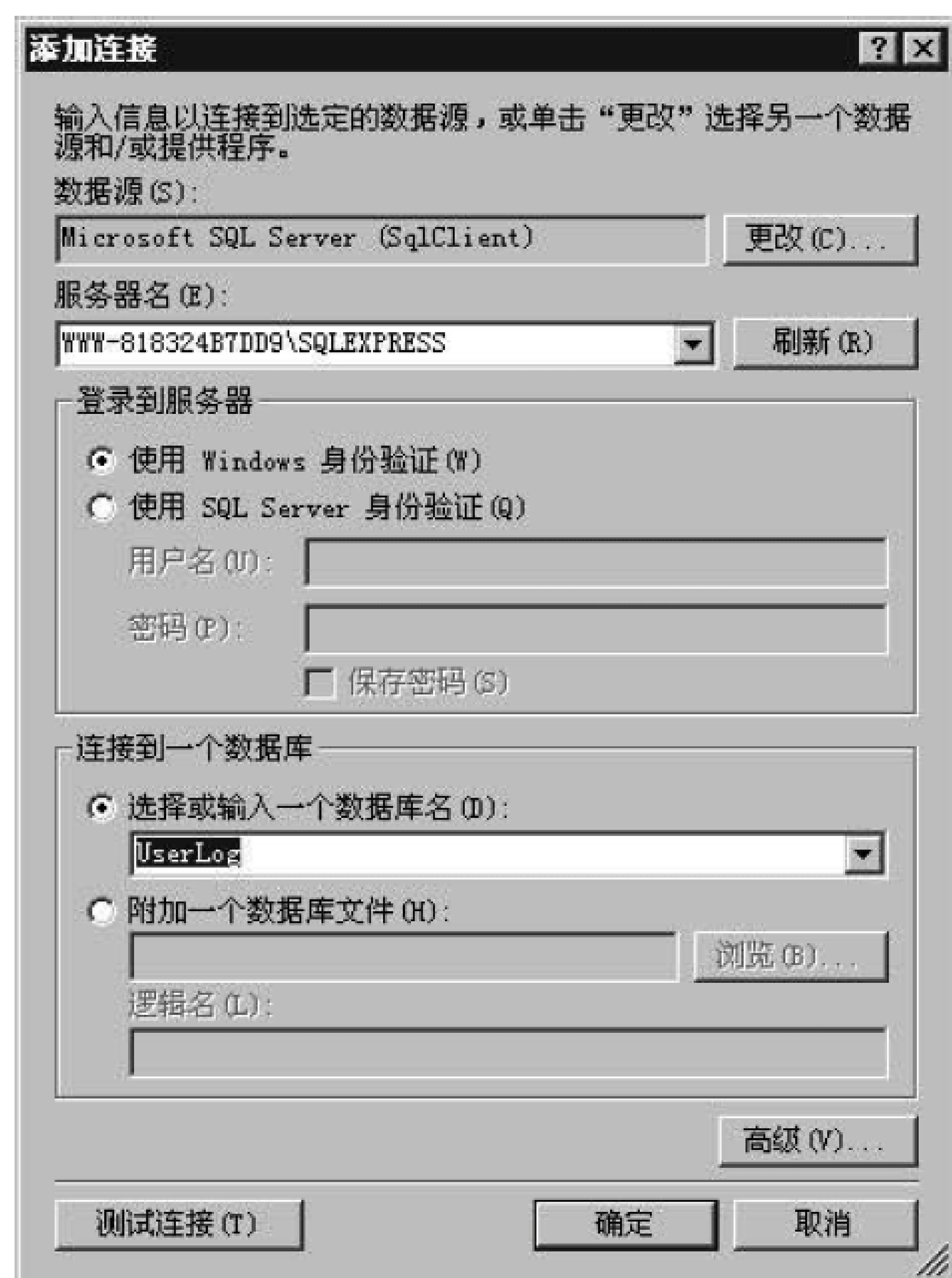


图 9-11　“添加连接”对话框



图 9-12　“更改数据源”对话框

（4）在图 9-11 中，在“服务器名”下拉列表框中选择数服务器名称，当然也可以通过“刷新”按钮自动获取当前网络中可用的 SQL Server 服务器，包括 SQL Server 2000、SQL Server 2005 和 SQL Server 2008。在“数据源”文本框中输入要连接的数据库名称，会自动列出可用的数据库。

（5）单击“测试连接”按钮测试是否能正常连接到数据库，通过“高级”按钮可以查看当前数据库连接的详细属性。最后通过“确定”按钮添加连接。

这样在“服务器资源管理器”窗体中的“数据连接”结点下，就会列出刚才增加的 Access

数据库连接，它的结点用数据库文件名表示。

9.3.3　用 Visual Studio 2010 管理数据库

将数据库连接到 Visual Studio 2010 之后，就可以通过它管理数据库。根据不同的数据源，在管理能力上有所差异，比如 SQL Server 数据源可以创建表、设计表等，而 Access 数据库则不能。但是无论它所支持的功能有多少，在操作界面和功能定义上都和 SQL Server Management Studio 十分相似，因为 Visual Studio 2010 本身就是通过 SQL Server 提供的接口来操作数据库，在这里就不再赘述。

9.4　小　　结

本章首先介绍了 SQL Server 2008 数据库的安装过程，然后介绍它的功能组件和平台结构，读者对它应该具有一个整体上的了解。重点介绍 SQL Server 2008 的数据库管理功能，介绍 SQL Server Management Studio 的使用。通过本章的学习，读者应该掌握以下知识点：

- ❑ SQL Server 2008 具有哪些功能组件？
- ❑ 如何通过 SQL Server Management Studio 新建数据库？
- ❑ 如何通过 SQL Server Management Studio 设计数据表？
- ❑ 如何通过 SQL Server Management Studio 修改数据记录？
- ❑ 如何通过 SQL Server Management Studio 设计数据库关系？
- ❑ 如何通过 SQL Server Management Studio 设计视图？
- ❑ 如何通过 Visual Studio 2010 创建 SQL Server 数据库？
- ❑ 如何通过 Visual Studio 2010 连接到数据库？

第 10 章　Transact-SQL 语言

虽然 SQL Server Management Studio 提供了大部分 SQL Server 数据库管理功能，但是在批量操作、复杂操作上，它并不十分方便。SQL Server 2008 支持 Transact-SQL 语言，通过该语言可以实现任何数据库的管理功能，包括很多 SQL Server Management Studio 不能实现的功能。本章将介绍 Transact-SQL 的具体使用。

10.1　T-SQL 简介

Transact-SQL 是微软实现的标准 SQL 语言，用于编写代码对 SQL Server 数据库进行管理，可以实现任何 SQL Server Management Studio 能实现的功能，而且可以通过 Visual Studio 2010 创建数据库项目进行开发。

10.1.1　什么是 T-SQL

SQL 是英文 Structured Query Language 的简称，译为结构化查询语言，由于它接近于英语口语，简洁易学，功能丰富，使用灵活，受到广泛的支持。经不断发展完善和扩充，SQL 被美国国家标准学会（ANSI）确定为关系型数据库语言的美国标准，后又被国际标准化组织（ISO）采纳为关系型数据库语言的国际标准。SQL 语言具有以下特点：

- 一体化：SQL 虽然称为结构化查询语言，但实际上它可以实现数据查询、定义、操纵和控制等全部功能。它把关系型数据库的数据定义语言 DDL、数据操纵语言 DML 和数据控制语言 DCL 集为一体，统一在一个语言中。
- 高度非过程化：用 SQL 语言进行数据操作，只需指出“做什么”，无须指明“怎么做”，存取路径的选择和操作的执行是由数据库管理系统（DBMS）自动完成。
- 两种使用方式和统一的语法结构：SQL 语言既是自含式语言，又是嵌人式语言。作为自含式语言，它可单独使用，用户在终端上直接键入 SQL 命令就可以实现对数据库进行操作。

如今，所有的数据库生产厂家都推出了各自的支持 SQL 的数据库管理系统，如微软的 SQL Server、IBM 的 DB2、Oracle、Sybase、Informix 等，也各自拥有对于标准 SQL 语言的具体实现。

T-SQL 的全称是 Transact-SQL，它是使用 SQL Server 的核心。任何与 SQL Server 实例通信的应用程序都通过将 Transact-SQL 语句发送到服务器(不考虑应用程序的用户界面）来实现对数据库的各种操作。通过 T-SQL 可以创建和管理 SQL Server 数据库对象，以及

插入、检索、修改和删除数据等，还可设置数据库的用户权限等信息。

T-SQL 作为一种数据库编程语言，同样具有变量、访问域、函数等概念，通过它可以编写出非常复杂的数据库存储过程，存储过程也是大型应用软件常用的数据库操作技术。本章将介绍 T-SQL 的更多使用细节。

10.1.2　创建 Visual Studio 2010 数据库项目

在 Visual Studio 2010 中，可以创建一种类型为“数据库项目”应用程序项目，在这里面可以编写并执行 SQL 脚本（*.sql）和数据库查询脚本（*.dqt），并从“输出”窗口打印出脚本执行情况。可以通过以下几个步骤创建数据库项目。

（1）打开 Visual Studio 2010，通过菜单“开始”|“新建”|“项目”打开“新建项目”对话框。在其中选择“数据库项目”模板，输入项目名称，单击“确定”按钮创建项目，这里项目名为 UserLogDbPrj。

（2）在弹出的“添加数据库引用”对话框中选择数据库 UserLog，如果没有，则通过“添加新引用”按钮添加 UserLog 数据库，如图 10-1 所示。

（3）在“解决方案管理器”中选中项目结点 UserLogDbPrj 右击，在弹出的快捷菜单中选择“添加 SQL 脚本”选项，弹出如图 10-2 所示的“添加新项”对话框。可以看出，可以添加多种类型的 SQL 脚本，各种类型都会产生一些指导性描述，本章后面将逐步使用它们。



图 10-1　添加数据库引用对话框



图 10-2　添加数据库引用对话框

（4）在图 10-2 中，选择模板“SQL 脚本”取名为 SelectUsr，单击“确定”按钮添加脚本 SelectUsr.sql 到项目 UserLogDbPrj 中，并编辑 SelectUsr.sql 的内容如下：

```
select * from User;
```

（5）在“解决方案资源管理器”中选中 SelectUsr.sql，右击，在弹出的快捷菜单选择“运行”选项，在“输出”|“数据库输出”窗口可以看到该脚本的输出情况。

提示：在 Visual Studio 2010 中，还有一个类型为“SQL Server 项目”的应用程序项目和数据库项目具有类似功能，有兴趣的读者可以自己试验一下。

10.2　通过 T-SQL 管理数据库

T-SQL 是微软为了让 SQL Server 数据库管理和交互更加容易，而在 SQL 标准基础上扩展而来的一种数据库管理语言，本节将介绍通过 T-SQL 如何实现常用的数据库管理功能。

10.2.1　用 CREATE DATABASE 创建数据库

T-SQL 提供了一系列的 CREATE 命令，顾名思义，这一组命令用来创建数据库中的各种对象，包括数据库、表、关系、视图等。表 10-1 列出了最常用的 CREATE 命令及其功能，本节将介绍 CREATE DATABASE 命令。

表 10-1　T-SQL中常用的CREATE命令

命　　令	功　　能
CREATE DATABASE	创建一个数据库及其存储数据的文件，或者附加一个已有的数据库文件到数据库服务器
CREATE TABLE	为指定的数据库创建一个新的表
CREATE INDEX	为指定的表或视图创建关系索引
CREATE PROCEDURE	为指定的数据库创建一个存储过程
CREATE VIEW	为指定的数据库创建一个视图（又叫虚拟表）
CREATE USER	为指定的数据库创建一个新用户
CREATE LOGIN	为指定的数据库创建新的SQL Server登录名
CREATE TYPE	创建一个自定义的数据类型
CREATE FUNCTION	创建一个自定义的函数

在 T-SQL 中，可以通过 CREATE DATABASE 命令创建一个新的数据库，该命令的格式如下所示。其中 database_name 表示新数据库的名称，是必需的参数，而且不能与数据库服务器中已有的数据库重名。ON 参数用来显示指定存储数据库数据的文件和文件组，PRIMARY 表示存储数据的文件，LOG ON 则表示存储日志的文件。COLLATE 指定数据库采用的默认排序方式。

```
CREATE DATABASE database_name
[ ON
[ PRIMARY ] [ <filespec> [ ,...n ]
[ , <filegroup> [ ,...n ] ]
[ LOG ON { <filespec> [ ,...n ] } ]
]
[ COLLATE collation name ]
[ WITH <external access option> ]
][;]
```

注意：在本章的命令格式定义代码中，“[ ]”内的参数是可选参数，“< >”内的元素是一个表示定义单元的标签，通常具有更详细的定义。另外，T-SQL 中，每个语句后面的分号“;”都是可选的，在只有单个语句时通常不写。

在这里，采用 CREATE DATABASE 最基本的格式（即只指定数据库名）创建一个名为 UserLog2，和第 9 章的数据库 UserLog 具有相同表结构的数据库。通过以下几个步骤完成这个操作：

（1）打开 10.1.2 节创建的数据库项目 UserLogDbPrj，在“解决方案资源管理器”中选中结点 UserLogDbPrj，在右键快捷菜单中选则“添加 SQL 脚本”选项。

（2）添加新的 SQL 脚本，取名为 CreateDB.sql，并编写如下 T-SQL 代码，如示例代码 10-1 所示。

示例代码 10-1

```
USE [master]
go
CREATE DATABASE UserLog2;
```

（3）选中脚本 CreateDB.sql，并通过右键快捷菜单中的“运行”或“运行于”选项（如图 10-3 所示）执行该脚本，从“输出”|“数据库输出”窗口中可以看到执行结果。

（4）如果“数据库输出”窗口的提示信息中没有失败提示，那么证明执行成功，可以通过 SQL Server Management Studio 查看新创建的数据库 UserLog2。



图 10-3　SQL 脚本执行命令

技巧： 如果执行该脚本得到失败提示类似于“无法获得数据库'model'上的排他锁”，通常是因为 Model 数据库正在被使用中。只需停止使用 Model 数据库，并关闭所有相关的数据库链接即可。

除了创建新数据库以外，CREATE DATABASE 命令还可以用来创建数据库快照，也可以将一个已有的数据库文件（*.mdf）文件附加到数据库服务器。限于篇幅，本书就不做介绍了，有兴趣的读者可以查看 SQL Server 联机帮助或相关书籍。

10.2.2　用 CREATE TABLE 创建数据表

在 T-SQL 中，可以通过 CREATE TABLE 命令为某个数据库创建数据库表，它的定义如下所示。其中必需的参数有如下 3 个。

- database_name：表示要创建表的数据库名，它必须是当前连接能够访问的数据库之一。
- table_name：表示新建表的名称。
- column_definition：表示新建表中字段（列）的信息，包括名称、类型等。

```
CREATE TABLE
[database name.[schema name].|schema name.]table name
({<column definition>|<computed column definition>}
   [<table constraint>][,...n])
   [ON{partition scheme name(partition column name)|filegroup
      |"default"}]
   [{TEXTIMAGE ON{filegroup|"default"}]
[;]
```

在创建数据库表时，必须同时指定它的字段信息，即 column_definition 参数，该参数及 data_type 参数更详细的格式定义及其中各参数的含义如下。

- column_name：表示字段的名称，在同一个数据表中字段名不能重复。
- NUL | NOT NULL：表示该字段是否允许为空，NULL 表示允许为空。
- colation_name：表示字段的排序规则。
- constraint_name：表示字段的约束名，在同一个数据库架构中不能重复。
- type_name：表示数据类型的名称，通过后面的括号“( )”可以指定长度、精度、最大值等信息。

```
<column definition>::=
column name<data type>
    [COLLATE collation_name]
    [NULL|NOT NULL]
    [
        [CONSTRAINT constraint_name]DEFAULT constant_expression]
      |[IDENTITY[(seed ,increment)][NOT FOR REPLICATION]
    ]
    [ROWGUIDCOL][<column constraint>[...n]]

<data type>::=
[type schema name.]type name
    [(precision[,scale]|max|
        [{CONTENT|DOCUMENT}]xml_schema_collection)]
```

从 CREATE TABLE 命令的定义可以看出，通过该命令可以为数据库表指定各种属性。本例将在 10.2.1 节创建的数据库 UserLogs2 上创建一个数据表 Users，T-SQL 代码如示例代码 10-2 所示。

示例代码 10-2

```
USE [UserLog2]
GO
CREATE TABLE[dbo].[Users](
    [LoginID][nvarchar](25) COLLATE Chinese_PRC_CI_AS NOT NULL,
    [Password][nvarchar](25) COLLATE Chinese PRC CI AS NOT NULL,
    [Name][nvarchar](50) COLLATE Chinese PRC CI AS NULL,
    [Age][int] NOT NULL,
    [XingBie][nvarchar](2)COLLATE Chinese_PRC_CI_AS NULL,
    [Mobile][nvarchar](20)COLLATE Chinese PRC CI AS NULL,
    [Email][nvarchar](100)COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT[PK Users]PRIMARY KEY CLUSTERED
    (
        [LoginID]ASC
    )WITH(IGNORE DUP KEY=OFF)ON[PRIMARY]
)ON[PRIMARY]
```

示例代码 10-2 中，数据库为 UserLog2，新的表名为 Users，包括 LoginID、Password、Name、Age、XingBie、Mobile、Email 共 7 个字段，每个字段都有数据类型、是否为空、排序规则等定义。比如字段 LoginID 的类型为字符类型 nvarchar，最大长度为 25，排序规则为 Chinese_PRC_CI（即简体中文），而且不能为空（NOT NULL）。同时，还通过 CONSTRAINT 关键字指定了数据表的约束——PK_Users，它指定表 Users 的主键（PRIMARY KEY）为字段 LoginID。

🔔注意：示例代码 10-2 中，dbo.Users 中的 dbo 是一个全局变量，表示当前使用的数据库，USE 子句指定当前要使用的数据库，这也是 T-SQL 中推荐的写法。

10.2.3　用 ALTER TABLE 创建数据库关系

在 T-SQL 中，创建数据表的关系也就是多个数据表之间的约束信息，可以通过 ALTER TABLE 命令来更改、添加和删除约束或字段信息，从而更新数据表及其关系。ALTER TABLE 命令用于添加数据表约束部分的格式定义如下：

```
ALTER TABLE[database name.[schema name].|schema name.]table name
{
   [WITH{CHECK|NOCHECK}]ADD
   {
      <table constraint>
   }[,...n]
}[;]

<table constraint>::=
[CONSTRAINT constraint name]
{
   {PRIMARY KEY|UNIQUE}
       [CLUSTERED|NONCLUSTERED]
          (column[ASC|DESC][,...n])
       [WITH FILLFACTOR=fillfactor
       [WITH(<index_option>[,...n] )]
       [ON{partition_scheme_name(partition_column_name...)
         |filegroup|"default"}]
   |FOREIGN KEY  (column[,...n])
       REFERENCES referenced table name[(ref column[,...n])]
       [ON DELETE{NO ACTION|CASCADE|SET NULL|SET DEFAULT}]
       [ON UPDATE{ NO ACTION|CASCADE|SET NULL|SET DEFAULT}]
       [NOT FOR REPLICATION]
}
```

其中，ADD 表示添加，table_constraint 表示添加约束。约束可以是主键（PRIMARY KEY）或者外键（FOREIGN KEY），添加主键的代码在 10.2.2 节其实已经使用过。作为数据表关系的约束通常是外键关系，本节将介绍外键。

在创建数据表关系之前，首先需要在数据库 UserLog2 中创建一个新的数据库表 Logs，它用来保存用户日志记录信息。添加 Logs 的 SQL 脚本如示例代码 10-3 所示，Logs 表包括 4 个列，其中 ID 为主键（通过 CONSTRAINT 子句表明），UserID 是 nvarchar(25)类型，将作为外键与表 Users 的 LoginID 关联。

示例代码 10-3

```
USE UserLog2
GO
CREATE TABLE [dbo].[Logs](
    [ID] [int] NOT NULL,
    [UserID] [nvarchar](25) COLLATE Chinese PRC CI AS NOT NULL,
    [LogContent] [nvarchar](1024) COLLATE Chinese PRC CI AS NOT NULL,
    [LogTime] [datetime] NOT NULL,
CONSTRAINT [PK Logs] PRIMARY KEY CLUSTERED
(
```

```
    [ID] ASC
)WITH (IGNORE DUP KEY=OFF)ON[PRIMARY]
)ON[PRIMARY]
GO
```

注意：由于 Logs.UserID 要与 Users.LoginID 作为约束（关系）的主键和外键，所以 Logs.UserID 的数据类型必须与 Users.LoginID 一致，否则接下来创建关系会失败。

在创建数据表 Logs 之后，为 Logs 和 Users 创建关系，即创建一个约束：Logs.UserID 作为表 Logs 的外键，关联到 Users.LoginID 上。执行该操作需要通过 ALTER TABLE 命令执行，SQL 脚本如示例代码 10-4 所示。这里通过关键字 FOREIGN KEY 指明约束类型为外键，名称为 FK_Logs_Users，通过关键字 REFERENCES 指明该外键要引用的数据表及其主键。

示例代码 10-4

```
USE [UserLog2]
GO
ALTER TABLE [dbo].[Logs] WITH CHECK ADD CONSTRAINT [FK Logs Users] FOREIGN
KEY([UserID])
REFERENCES [dbo].[Users] ([LoginID])
```

技巧：SQL 脚本中的 T-SQL 语言是依次执行，所以可以同时在一个脚本中出现多个语句。也就是说可以将示例代码 10-3 和示例代码 10-4 合并为一个脚本，一次性执行。

10.2.4　用 INSERT 插入数据库记录

通过 T-SQL 同样可以为数据库中的表插入一条数据库记录，在 T-SQL 中通过 INSERT 命令添加一条数据库记录到表或视图，本节只介绍添加到表。INSERT 命令常用的定义如下：

```
[WITH<common table expression>[,...n]]
INSERT
[INTO]
{table name|rowset function limited
[WITH(<Table_Hint_Limited>[...n])]
}
{
[(column list)]
[<OUTPUT Clause>]
{VALUES({DEFAULT|NULL|expression}[,...n])
|derived table
|execute statement
}
}
|DEFAULT VALUES
[;]
```

这里列出的只是 INSERT 语句最常用的格式，但是已经足够满足常见的软件开发需要，其中各关键字所表示的含义如下：

- WITH：指定用特定的 SELECT 语句查询产生的数据作为 INSERT 语句的数据源。
- INSERT：表明是插入记录命令。
- INTO：指明要插入记录的数据表，给出表名即可。
- column_list：表示要插入记录的列名。
- VALUES：指明新记录各字段的值，依次填入 column_list 中各个字段的值。
- DEFAULT VALUES：表示强制要求新记录使用数据表中字段定义的默认值。

在 INSERT 命令中，如果不指定 column_list，则 VALUES 关键字中的值应该按照数据库表创建时指定的顺序列出。如示例代码 10-5 所示，通过 VALUES 中的所有列数据就按照 Users 创建时的顺序排列，依次为 LoginID、Password、Name、Age、XingBie、Mobile、Email。

示例代码 10-5

```
USE UserLog2
GO
INSERT  INTO Users
VALUES('User1', 'Password1', '李四', 20, '男', '13112345678', 'lisi@126.
com')
GO
```

在 T-SQL 中，文本类型（如 nvarchar、varchar、char、nchar 等）的值用单引号“'”括起来，如示例代码 10-5 中 LoginID 字段的值'User1'，而证书则直接用数字即可，如 Age 字段的值 20。

在实际开发中，由于开发人员通常不是数据库设计人员，也不知道数据库表字段的默认顺序，所以示例代码 10-5 中的写法容易出错。INSERT 语句还可以通过 column_list 用括号“( )”把多个字段名按照特定的排序排列，VALUES 中的字段值按照 column_list 指定的顺序排列。如示例代码 10-6 所示，按照指定的顺序为表 Users 添加一个新记录。

示例代码 10-6

```
USE UserLog2
GO
INSERT INTO [Users]([Name], [LoginID], [Password], [Age], [Email], [XingBie],
[Mobile])
VALUES('王五', 'User2', 'Password2', 30, 'wangwu@126.com', '男', '13645671234')
GO
```

技巧： 在 T-SQL 中如果数据库名、表名和字段名与 T-SQL 关键字相同时，可以通过中括号“[ ]”将该字段括起来表示它不是关键字，如示例代码 10-6 中的[Name]。最常用的办法是将所有字段都用中括号括起来，也可以清晰地表示 T-SQL 脚本中哪些是非关键字。

在一些数据表中，某些列允许数据为空或定义了默认值，在 INSERT 命令中可以不指定这些列的值，那样这些列的值相应地为空或默认值。如示例代码 10-7 所示的 SQL 脚本，插入 4 个新记录到表 Users，都只指定了 3 个必须（不能为空）的字段值。

示例代码 10-7

```
INSERT  INTO  [Users]([LoginID],  [Password],  [Age])
VALUES('User3', 'Password3', 21)
```

```
INSERT INTO [Users]([LoginID], [Password], [Age])
VALUES('User4', 'Password3', 22)
INSERT INTO [Users]([LoginID], [Password], [Age])
VALUES('User5', 'Password3', 23)
INSERT INTO [Users]([LoginID], [Password], [Age])
VALUES('User6', 'Password3', 24)
GO
```

注意：如果该某个字段既没有默认值，也不能为空，那么在 INSERT 命令中必须要指定该字段的值；否则 SQL 脚本将出现执行错误。

10.2.5　用 UPDATE 更新数据库记录

在 T-SQL 中，通过 UPDATE 命令更新数据库记录，它可以更新数据表中的所有记录，也可以更新满足指定条件的记录，UPDATE 命令的最常用的格式定义如下：

```
UPDATE table name
   SET { column name = { expression | DEFAULT | NULL }} [ ,...n ]
   [ WHERE { <search condition> } ]
[ ; ]
```

UPDATE 命令通常包括 UPDATE、SET、WHERE 3 个部分的参数。其中，UPDATE 表示该命令为更新数据库记录，table_name 指定要更新的数据表名。SET 参数为必须的，它指定要更新的字段已经更新的值。WHERE 子句通过一组具有逻辑值的表达式来表示需要更新的数据记录所满足的条件。

示例代码 10-8 是只有 FROM 子句的 UPDATE 命令，也是最简单的 UPDATE 命令，这里它将所有用户的年龄都增加 1 岁。其中，UPDATE Users 表示要更新数据库表 Users 的记录，SET 表示要更新 Users.Age，新的值为 Users.Age+1，即年龄加 1。

示例代码 10-8

```
USE UserLog2
GO
UPDATE Users
SET Users.Age = Users.Age+1
GO
```

注意：在 T-SQL 中同样可以包括加、减、乘、除、大于、小于等运算，还可以执行函数，自定义类型等，更多信息将在 10.3 节介绍。

由于示例代码 10-8 中，并没有使用 WHERE 子句对记录进行过滤，所以表 Users 中所有的记录都会受到更新。在实际开发中，通常只是需要更新满足某些条件的记录，这就需要通过 WHERE 子句对记录进行过滤。

WHERE 子句通常由一个或多个具有逻辑值的表达式组成，逻辑表达式之间通过并（AND）、或（OR）、非（NOT）3 个运算符进行逻辑运算，它们的功能如表 10-2 所示。

- NOT：对逻辑表达式求反，true 变 false，false 变 true。
- AND：组合两个逻辑表达式，并在两个表达式都为 true 时取值为 true。
- OR：组合两个逻辑表达式，并在任何一个表达式为 true 时取值为 true。

❑ 当 UPDATE 命令具有 WHERE 子句时，只有满足 WHERE 子句指定条件的记录才会执行更新操作。如示例代码 10-9，WHERE 子句“WHERE (Users.Age < 25) AND (Users.Age > 20)”表示 Age 在 20～25 之间的记录才会执行 SET 子句指定的运算。

示例代码 10-9

```
USE UserLog2
GO
UPDATE Users
SET Users.Age = Users.Age + 2
WHERE (Users.Age < 25) AND (Users.Age > 20)
GO
```

注意：和常见的编程语言一样，T-SQL 中的 WHERE 子句可以通过小括号“()”来改变逻辑运算的顺序，而且多个条件之间也应该通过“( )”运算符区分去运算顺序，使得代码更直观。

10.2.6　用 DELETE 删除数据库记录

在 T-SQL 中，通过 DELETE 命令删除数据库记录，和 UPDATE 命令一样，它可以删除数据表的所有记录，也可以通过 WHERE 子句来删除满足指定条件的记录。DELETE 命令常用的定义如下：

```
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM <table source> [ ,...n ] ]
    [ WHERE { <search condition> } ]
[; ]
```

其中，TOP 表示要删除记录的数量或百分比，如 TOP (10)表示删除前面 10 条记录。而 PERCENT 关键字则表示要删除的百分比，如 TOP (10) PERCENT 则表示记录总数的前面 10%。FROM 子句表示要删除的数据库表名，WHERE 子句则表示要删除的记录必须满足的条件。

示例代码 10-10 包含 3 个 DELETE 代码，其中第 1 个 DELETE 子句，由于有参数 TOP(1)，所以它将删除数据表 Logs 中的第 1 条记录。第 2 个 DELETE 子句，由于参数 TOP(10)，所以它将删除数据表 Logs 的前面 10%的记录。第 3 条 WHERE 子句，由于没有任何限制条件，所以该代码将删除表 Logs 中所有的记录。

示例代码 10-10

```
USE UserLog2
GO
DELETE TOP(1) FROM Logs;
GO
DELETE TOP(10)PERCENT FROM Logs;
GO
DELETE FROM Logs;
```

注意：如果数据库表中已经没有任何记录，将不做任何操作。如果数据库表中的记录数不能满足指定的删除数量，则将所有记录删除。

在 DELETE 命令中，通过 WHERE 子句对数据表中的记录进行过滤，只有满足指定条件的记录才会被删除。如示例代码 10-11 所示，共 3 个 DELETE 子句，都是删除满足条件 Users.Age > 20 的记录，不满足条件的记录不会受到影响。

示例代码 10-11

```
USE UserLog2
GO
DELETE TOP(1)
FROM Users
WHERE Users.Age > 20;
GO
DELETE TOP(10)PERCENT
FROM Users
WHERE Users.Age > 20;
GO
DELETE
FROM Users
WHERE Users.Age > 20;
GO
```

注意：当 DELETE 中存在 WHERE 子句时，TOP 参数是对满足 WHERE 子句条件的记录进行数量计算，而不是对所有记录进行数量计算。

10.2.7　用 SELECT 查询单表的记录

SELECT 命令是最常用的 SQL 命令之一，它用来查询数据库中的记录。在 T-SQL 中，SELECT 是一个相对复杂的命令，它具有 SELECT、FROM、WHERE、ORDER BY、GROUP BY、HAVING 等子句，分别表示不同的功能。既可以查询单个表的数据，也可以查询多个表的数据，可以指定查询条件，也可以指定返回记录数量等。

在 T-SQL 中，SELECT 命令的主要部分是 SELECT 子句，其格式如下：

```
SELECT [ ALL | DISTINCT ]
[ TOP expression [ PERCENT ] [ WITH TIES ] ]
<select list>
<select list> ::=
   {
     *
     | { table_name | view_name | table_alias }.*
     | { column name | [ ] expression | $IDENTITY | $ROWGUID }
     [ [ AS ] column alias ]
     | column alias = expression
   } [ ,...n ]
```

其中，select_list 是指定了要作为查询结果返回的列名称，星号（*）表示所有列都要返回，如 Users.*表示返回 Users 表中的所有列。AS 关键字指定该列在查询结果中的新名称，如果不指定则默认为数据表中的列名称。如 Users.LoginID AS LoginName，则

LoginName 是新的列名。多个返回列之间用逗号（,）分隔。

另外，在 SELECT 命令中必须通过 FROM 子句指定被查询的数据表名，可以是一个数据表，也可以是多个数据表，多个数据表之间用逗号（,）分隔。如示例代码 10-12 所示，其中第 1 个 SELECT 命令查询数据表 Users（FROM 子句指定），并返回 Users 中的所有记录，包括所有列（Users.*指定）的信息。第 2 个 SELECT 命令同样是查询表 Users，但是只返回 LoginID 和 Password 两个列的数据，并将 Password 列重命名为“密码”。

示例代码 10-12

```
USE UserLog2
GO
SELECT Users.*
FROM Users
GO
SELECT LoginID, Password AS 密码
FROM Users
GO
```

示例代码 10-12 中第 1 个 SELECT 命令的查询结果如图 10-4 所示，可见表 Users 的所有列都被返回，而且名称默认为列的名称。第 2 个 SELECT 命令的查询结果如图 10-5 所示，可见只有两列返回，而且 Password 列的名称为“密码”。

	LoginID	Password	Name	Age	XingBie	Mobile	Email
▶	User1	Password1	李四	24	男	13112345678	lisi@126.com
	User2	Password2	王五	32	男	13645671234	wangwu@126.com
	User3	Password3	NULL	24	NULL	NULL	NULL
	User4	Password3	NULL	26	NULL	NULL	NULL
	User5	Password3	NULL	25	NULL	NULL	NULL
	User6	Password3	NULL	26	NULL	NULL	NULL

图 10-4　SELECT 命令结果 1

	LoginID	密码
▶	User1	Password1
	User2	Password2
	User3	Password3
	User4	Password3
	User5	Password3
	User6	Password3

图 10-5　SELECT 命令结果 2

在 SELECT 命令中，同一个 FROM 子句包含多个数据表名，从而同时将多个表的记录作为结果返回，各表之间的记录连接。如示例代码 10-13 所示，这个 SELECT 命令查询表 Users 和 Logs 中的所有记录，输出如图 10-6 所示。

示例代码 10-13

```
USE UserLog2
GO
```

```
SELECT Users.LoginID, Users.Name, Logs.LogTime, Logs.LogContent
FROM Users, Logs
GO
```

	LoginID	Name	LogTime	LogContent
▶	User1	李四	2008-1-1 0:0...	User1's Log
	User2	王五	2008-1-1 0:0...	User1's Log
	User3	NULL	2008-1-1 0:0...	User1's Log
	User4	NULL	2008-1-1 0:0...	User1's Log
	User5	NULL	2008-1-1 0:0...	User1's Log
	User6	NULL	2008-1-1 0:0...	User1's Log
	User1	李四	2008-1-2 0:0...	User2's Log
	User2	王五	2008-1-2 0:0...	User2's Log
	User3	NULL	2008-1-2 0:0...	User2's Log
	User4	NULL	2008-1-2 0:0...	User2's Log
	User5	NULL	2008-1-2 0:0...	User2's Log
	User6	NULL	2008-1-2 0:0...	User2's Log

图 10-6　SELECT 命令结果 3

技巧： FROM 子句中的数据表名也可以通过 AS 关键字进行重命名，这样可以防止对一个表同时多次查询带来的表名混乱。如 FROM Users AS U1、Users AS U2，U1 和 U2 就可以区分两个表。

10.2.8　用 WHERE 查询指定条件的记录

很多时候，示例代码 10-13 的查询结果实际上没有实际意义，而且也不能满足开发需要。为了让查询结果变得更有意义，通常需要通过 WHERE 子句以一组具有逻辑值的表达式来表示需要更新的数据记录所满足的条件。

如示例代码 10-14 所示，该 SELECT 命令从表 Users 和 Logs 中查询所有用户的所有日志记录，WHERE 子句“WHERE Users.LoginID=Logs.UserID”指定该条件。该示例代码的输出如图 10-7 所示，从中可以看出，和用户无关的记录已经被过滤。

示例代码 10-14

```
USE UserLog2
GO
SELECT Users.LoginID, Users.Name, Logs.LogTime, Logs.LogContent
FROM Users, Logs
WHERE Users.LoginID=Logs.UserID
GO
```

	LoginID	Name	LogTime	LogContent
▶	User1	李四	2008-1-1 0:0...	User1's Log
	User2	王五	2008-1-2 0:0...	User2's Log

图 10-7　SELECT 命令结果 4

提示：SELECT 命令还可以通过 GROUP BY 子句对查询结果进行分组，比如示例代码 10-14 中按照 LoginID 分组，也可以通过 ORDER BY 子句对查询结果按照指定的顺序进行排序。

10.3　使用 T-SQL 高级特性

在 10.2 节介绍的命令都是 T-SQL 语句中最常见的命令，而且都是静态计算。T-SQL 作为一种开发语言，同样具有数据类型、运算符、函数等高级特性，本节将详细介绍这些内容。

10.3.1　常用的 T-SQL 数据类型

在 T-SQL 中，每个列、局部变量、表达式和参数都具有一个相关的数据类型。数据类型是一种属性，用于指定对象可保存的数据类型，最基础的数据类型主要有精确数字、近似数字、日期和时间、ASCII 字符串、Unicode 字符串、二进制字符串等。T-SQL 中的数据类型如表 10-2 所示。

表 10-2　T-SQL中常用的数据类型

类　别	数据类型	说　明
精确数字	bigint	8字节的整数，-2 E 63到2 E 63-1
	Int	4字节的整数，-2 E 31到2 E 31-1
	smallint	2字节的整数，-2 E 15到2 E 15-1
	tinyint	1字节的整数，0到255
	bit	存储1位（bit）的整数，只能为0、1或NULL
	decimal	带固定精度和小数位数的数字，格式为decimal(p,s)，p为精度，s为小数位数，都可选
	numeric	等价于decimal
	money	代表货币或货币值，8字节数据，范围为–922 337 203 685 477.5808～922 337 203 685 477.5807
	smallmoney	代表货币或货币值，8字节数据，范围为–214 748.3648～214 748.3647
近似数字	float	定义指定存储长度的小数，格式为float(n)，n为存储长度。n为1～24时，存储4字节数据，精度为7位数。n为25～53时，存储8字节数据，精度为15位数
	real	等价于float(24)
日期和时间	datetime	表示某天的日期和时间，1753-01-01到9999-12-31，精度为3.33毫秒
	smalldatetime	表示某天的日期和时间，1900-01-01到2079-06-06，精度为1分钟
字符串	char	固定长度的ASCII字符串，格式为char(n)，n表示字符串长度，最大为8000
	varchar	可变长度的ASCII字符串，格式为varchar(n)，n表示字符串最大长度，最大为8000
	text	长度可变的ASCII字符串数据，最大长度为2 E 31 – 1

续表

类　别	数 据 类 型	说　明
Unicode字符串	nchar	固定长度的Unicode字符串，格式为char(n)，n表示字符串长度，最大为4000
	nvarchar	可变长度的Unicode字符串，格式为varchar(n)，n表示字符串最大长度，最大为4000
	ntext	长度可变的Unicode字符串数据，最大长度为2 E 30 – 1
二进制字符串	binary	固定长度的二进制数据，格式为binary(n)，n表示数据的字节数，最大为8000
	varbinary	可变长度的二进制数据，格式为varbinary(n)，n表示数据的最大字节数，最大为8000
	image	可变长度的二进制数据，最大长度为2 E 31 – 1字节
其他数据类型	Uniqueidentifier	16字节长度的GUID，可以唯一标识一条记录
	xml	存储XML数据的数据类型
	table	用于临时存储数据集（如查询结果）的数据类型
	sql_variant	可以存储任意类型的数据类型

注意： 表 10-2 数字中的 E 表示幂运算，如 2 E 10 表示 2 的 10 次方。T-SQL 中的数据类型和 SQL Server 中的数据类型基本上是一致的。

T-SQL 的数字类型包括长度、精度和小数位数 3 个属性，长度是存储此数所占用的字节数，精度是数字中的数字个数，小数位数是数中小数点右边的数字个数。例如，int 数据类型的精度是 10，长度是 4，小数位数是 0。而小数 12345.45 的精度是 7，小数位数是 2。

字符串或 Unicode 数据类型的长度是字符个数，binary、varbinary 和 image 数据类型的长度是字节数。当两个 char、varchar、binary 和 varbinary 表达式串联时，所生成表达式的长度是两个源表达式长度之和，最大为 8000 个字符。当两个 nchar 或 nvarchar 表达式串联时，所生成表达式的长度是两个源表达式长度之和，最大为 4000 个字符。

除了 decimal 类型之外，数字数据类型的精度和小数位数是固定的。如果算术运算符有两个相同类型的表达式，结果就为该数据类型，并且具有对此类型定义的精度和小数位数。如果运算符有两个不同数字数据类型的表达式，则由数据类型优先级决定结果的数据类型。结果具有为该数据类型定义的精度和小数位数。

10.3.2　常用的 T-SQL 运算符

在 T-SQL 中，运算符是一种符号，用于指定要在一个或多个表达式之间执行的操作。和其他编程语言一样，T-SQL 也包含多种类型的运算符，如表 10-3 所示。

表 10-3　T-SQL中常用的运算符

类　别	运　算　符	说　明
算术运算符	+（加）	对两个数值类型和时间类型数据进行加法运算，也可以对字符串类型执行联接运算
	-（减）	对两个数值类型和时间类型数据进行减法运算
	*（乘）	对两个数值类型数据进行乘法运算
	/（除）	对两个数值类型数据进行除法运算
	%（模）	对两个整数类型数据进行求模运算，即返回整数余数
	+（正）	表示一个数值类型为正数
	-（负）	表示一个数值类型取相反数。如-5，-(-5)等于+5

续表

类　别	运　算　符	说　明
赋值运算符	=（等号）	等号是T-SQL中唯一的赋值运算符，如SET @MyVar=1.2
按位运算符	&（位与）	对整数类型或二进制类型数据执行按位与的运算，当对应的两位都为1时，该位为1，否则该位为0
	|（位或）	对整数类型或二进制类型数据执行按位或的运算，当对应的两位都为0时，该位为0，否则该位为1
	^（位异或）	对整数类型或二进制类型数据执行按位异或的运算，当对应的两位相同时，该位为0，否则该位为1
	~（位非）	对整数类型或二进制类型数据执行按位取反的运算，当对应位为1时，该位为0，否则为1
比较运算符	=（等于）	比较两个操作数是否相等
	>（大于）	比较左操作数是否大于右操作数
	<（小于）	比较左操作数是否小于右操作数
	>=（大于等于）	比较左操作数是否大于等于右操作数
	<=（小于等于）	比较左操作数是否小于等于右操作数
	<>（不等于）	比较两个操作数是否不相等
	!=（不等于）	等价于<>
	!<（不小于）	比较左操作数是否不小于右操作数
	!>（不大于）	比较左操作数是否不大于右操作数
逻辑运算符	ALL	如果对一组数据的条件判断（如比较运算）都成立，那么ALL表达式也成立，返回true，否则返回false
	ANY	如果一组数据中任意一个满足条件，那么ANY表达式成立，返回true；否则返回false
	BETWEEN…AND	如果操作数在某个范围之内，返回true；否则返回false。如@IntVal BETWEEN 1 AND 10
	IN	如果操作数等于给出数据集合中的任意一个，则成立，返回true；否则返回false
	EXISTS	如果子查询中包含一些行，那么返回true；否则返回false
	LIKE	如果操作数与某个模式（正则表达式）匹配，则返回true；否则返回false
	AND	如果两个布尔表达式都为true，则返回true；否则返回false
	NOT	对布尔表达式取反，如果表达式为true，则返回false
	OR	如果两个布尔表达式都为false，则返回false；否则返回true
	SOME	如果一组布尔表达式中有一个或多个为true，则返回true；否则返回false

从表 10-3 中可以看出，T-SQL 中的运算符和平常的编程语言（如 C#）相比更多一些，尤其是逻辑运算符，这主要是因为作为查询语言，T-SQL 要求更多的方便性和灵活性，而且对数据集合的支持要求更加强大和多样化。

T-SQL 中逻辑运算符比其他编程语言的逻辑运算符更加灵活，比如 ALL、ANY、BETWEEN…AND、IN、EXISTS 等，它们可以非常简单地描述一些常见但编程语言少有的操作。如@IntVal BETWEEN 1 AND 10，表示变量 IntVal 的值要求在 1～10 之间，如果成立返回 true，不成立则返回 false。比如 WHERE @StrVal LIKE 'ABC'，表示如果变量 StrVal 的值包含'ABC'字符，则返回 true；否则返回 false。

另外，大部分 T-SQL 的运算符不仅对一个或多个变量进行运算，最主要是对整个数据集（即一组数据）进行运算，它们与 SELECT、WHERE 等语句集成到一起，可以开发出非常高效和灵活的数据库查询代码。

🔔注意：T-SQL 的重点在于数据集合和数据记录的处理，而不是逻辑功能的运算，不要把它作为 C#这样的语言来使用，可以通过 CAST 进行强制类型转换。

10.3.3　使用 T-SQL 表达式

前面介绍了 T-SQL 中的数据类型和运算符，将这些数据类型和运算符组合到一起就成为了表达式。T-SQL 表达式也具有数据类型，表达式的数据类型依赖于参与运算的运算符和操作数，如“1+2”的类型为整数，值为 3。“1.05+3.03”的类型为小数，值为 4.08。而“'ABC'+'HELLO'”的类型为字符串，值为'ABCHELLO'。比较运算符和逻辑运算符产生的表达式为布尔类型，如果逻辑成立就为 true，逻辑不成立就为 false。如“1 > 3”为布尔类型，值为 false。而“1 <= 4”的值为 true。

在 T-SQL 中，最简单的表达式为常量，就前面例子中的 1.05、3.03、'ABC'等，它们的值是固定不变的，直接在代码里面编写。变量也是一种很简单的表达式，它用一个符号来唯一表示一个值可以改变但固定类型的表达式，可以通过 DECLARE 声明一个变量，然后用 SET 或 SELECT 为该变量赋值。

如示例代码 10-15 所示，通过 DECLARE 声明一个变量 userid，类型为 varchar(10)，通过 SET 设置它的值为'User1%'，最后在 SELECT 命令的 WHERE 子句中使用它。

示例代码 10-15

```
DECLARE @userid varchar(10);
SET @userid = 'User1%';
SELECT Users.LoginID, Users.Name, Users.Age
FROM Users
WHERE Users.LoginID LIKE @userid
GO
```

示例代码 10-15 为查询数据表 Users 中所有 LoginID 以 User1 开始的记录，产生的查询结果如图 10-8 所示，从中可以看出只有 LoginID 以 User1 前导的记录被查询出去，其他的记录如 User2、User3 等都没有查询出来。

	LoginID	Name	Age
	User1	李四	24
	User12	张三	24

图 10-8　SELECT 命令结果 5

🔔注意：在 T-SQL 中变量通过@符号来前导，表示它是一个变量，不是固定值或关键字。为变量赋值使用赋值运算符“=”。

在 T-SQL 中，只要两个表达式是某个运算符支持的数据类型，并且满足至少下列条件中的一个，即可以由这个运算符组合起来，成为一个新的表达式，如示例代码 10-15 那样。

- ❑ 两个表达式有相同的数据类型。
- ❑ 优先级低的数据类型可以隐式转换为优先级高的数据类型。
- ❑ CAST 函数能够显式地将优先级低的数据类型转化成优先级高的数据类型，或者转换为一种可以隐式地转化成优先级高的数据类型的过渡数据类型。

在计算表达式的值时，遇到多个运算符时需要按照运算符的优先级高低先后执行，通过圆括号“()”可以改变运算符的计算顺序。如，“1+2*3”中先算 2*3 得 6，后算 1+6 得 7，即表达式的值为 7，因为“*”的优先级要高于“+”。但是在“（1+2）*3”中，先算（1+2）得 3，后算 3*3 得 9，即表达式的值为 9，因为“()”优先级最高，它要求先计算 1+2。

注意：在 T-SQL 中，布尔类型除了 true 和 false 两个值外，还包括一个 UNKNOWN，表示该运算还没有执行，也没有产生任何结果。

10.3.4　使用 T-SQL 结构语句

在 T-SQL 中同样包含改变程序结构的语句，主要包括 IF…ELSE 和 WHILE 两个。IF…ELSE 在 T-SQL 中用于条件跳转，它的定义如下：

```
IF Boolean expression
   {sql statement|statement block}
[ELSE
   {sql_statement|statement_block}]
```

其中，Boolean_expression 是 IF 语句的判断条件，如果为 true，则执行 IF 子句中的代码块，否则执行 ELSE 子句的代码块。和其他开发语言类似，T-SQL 中的 IF 子句也可以不包括 ELSE 子句。另外，T-SQL 中的代码块，可以是普通的 T-SQL 代码，也可以是 SQL 查询代码。

如示例代码 10-16 所示，首先声明一个 int 类型变量 tableid，并设置它的值为 1，然后通过 IF…ELSE 语句选择要执行的查询代码。在该例中，由于@tableid 的值为 1，所以会执行 IF 子句对应的代码块，即查询表 Users 的数据记录。

示例代码 10-16

```
DECLARE @tableid int;
SET @tableid = 1
IF @tableid = 1
BEGIN
    SELECT Users.LoginID, Users.Name, Users.Age
    FROM Users
END
ELSE
BEGIN
    SELECT Logs.*
    FROM Logs
END
```

注意：T-SQL 中的代码块用 BEGIN 和 END 关键字包括起来，BEGIN 表示代码块开始，END 表示代码块结束，如示例代码 10-16 所示。

循环语句是开发语言中另外一个常用的语句，在 T-SQL 中，通过 WHILE 语句循环执行代码块。WHILE 语句的格式如下：

```
WHILE Boolean expression
    {sql statement|statement block}
    [BREAK]
    {sql_statement|statement_block}
    [CONTINUE]
    {sql_statement|statement_block}
```

其中，Boolean_expression 是 WHILE 语句是否继续执行的判断条件，如果为 true，则执行 WHILE 子句中的代码块，否则退出循环语句。在 T-SQL 的 WHILE 代码块中，可以通过 BREAK 中止整个循环，也可以通过 CONTINUE 中止本次循环，马上执行下一次循环。

10.3.5　使用 T-SQL 聚合函数

T-SQL 中包含很多内置函数，同时也可以创建自定义函数，自定义函数通常作为存储过程的一部分，将在 10.4 节介绍存储过程，本节重点介绍 T-SQL 内置的函数。T-SQL 中的函数按照功能分成以下几类。

- 行集函数：这类函数返回的对象通常是数据的集合，可以像数据表一样在 T-SQL 中被查询和使用，主要包括 CONTAINSTABLE、FREETEXTTABLE、OPENDATASOURCE 等。
- 聚合函数：聚合函数是对一组值进行计算，并返回单个值，本节将详细介绍。
- 排名函数：排名函数为一组数据中的每一行返回一个排名值。根据所用的函数，某些行可能与其他行接收到相同的值，所以排名函数具有不确定性
- 矢量函数：对单一值进行运算，然后返回单一值，只要表达式有效，即可使用标量函数。在 10.3.6 节将详细介绍。

在 T-SQL 中，常见的聚合函数如表 10-4 所示，可以对一组数据进行多种运算。聚合函数通常用在 SELECT 命令的选择列表和 HAVING 命令中。

表 10-4　T-SQL中常用的聚合函数

函　　数	说　　明
AVG	返回一组数据的平均值，NULL将被忽略，可以指定是否计算重复值
CHECKSUM	返回按照表的某一行或一组表达式计算出来的校验和值。CHECKSUM用于生成哈希索引
COUNT	返回一组数据的项数，始终返回int数据类型值，可以指定是否计算重复值
COUNT_BIG	返回一组数据的项数，始终返回bigint数据类型值，可以指定是否计算重复值
MAX	返回一组数据中的最大值，NULL将被忽略
MIN	返回一组数据中的最小值，NULL将被忽略
SUM	返回一组数据中的数据的和，NULL将被忽略，可以指定是否计算重复值

在聚合函数中，星号（*）通常表示包括 NULL 在内的任意记录，关键字 ALL 通常表示除了 NULL 以外的所有记录，关键字 DISTINCT 则表示除了 NULL 之外的记录中，如果出现重复的值只取值一次进行计算。

如示例代码 10-17 所示，第 1 个查询通过星号（*）表示计算数据表 Logs 中所有记录（包括 NULL 和重复记录）的数量。第 2 个查询使用关键字 ALL，将返回表 Users 中所有

记录的数量，包括重复记录，但是不包括值为 NULL 的记录。第 3 个查询使用关键字 DISTINCT，将返回所有表 Logs 中非空且不重复记录的数量。

示例代码 10-17

```
SELECT COUNT(*)
FROM Logs
GO
SELECT COUNT(ALL Users.LoginID)
FROM Users
GO
SELECT COUNT(DISTINCT Logs.UserID)
FROM Logs
GO
```

注意：在 T-SQL 的聚合函数中，ALL、DISTINCT 等关键字可以用于 AVG、SUM、COUNT、COUNT_BIG 等多个函数中，但功能都是相似的。

10.3.6　使用 T-SQL 数学计算函数

矢量函数也是 T-SQL 中常用的函数类型之一，但是矢量函数的数量很多，主要包括如表 10-5 所示的几种，从中可以看出 T-SQL 的确具有非常强大的功能，可执行多种不同的运算。

表 10-5　T-SQL中矢量函数的类型

函 数 类 型	说　　明
数学函数	对数值型（如int、bigint、float等）数据进行数学运算，包括绝对值、三角运算、随机数等
时间日期函数	对时间类型（SmallDatetime，DateTime）数据进行运算，返回字符串、数值或时间类型数据。包括获取年、月、日等
字符串函数	对字符串执行一些常见的操作，包括计算长度、截取左右空白、逆序等
元数据函数	返回数据库和数据库对象相关的信息
安全函数	返回用户和角色等安全相关的信息
配置函数	返回当前SQL Server数据库的一些配置信息
游标函数	返回当前SQL Server数据库的游标信息
系统函数	返回当前SQL Server数据库管理器的一些系统信息
系统统计函数	返回当前SQL Server数据库管理器的一些系统统计信息
文本和图像函数	对文本或图像输入数据进行处理，并返回相关信息

数学函数在实际的存储编写中也是常用的，它是 T-SQL 中进行数学运算的基础，提供了绝对值、三角运算、反三角运算、指数运算等，还支持随机数的生成。表 10-6 列出了 T-SQL 所支持的数学函数及其使用格式。

表 10-6　T-SQL中常见的数学函数

函　数　名	返 回 类 型	格　　式	说　　明
PI	float	PI	返回圆周率常量
ABS	任意数值类型	ABS(numeric_expression)	计算数值的绝对值
SQRT	float	SQRT(numeric_expression)	计算数值的平方根
SQUARE	float	SQUARE(numeric_expression)	计算数值的平方
POWER	float	POWER(numeric_expression, y)	计算数值的y次幂
EXP	float	EXP(numeric_expression)	计算e的numeric_expression次方
LOG	float	LOG(numeric_expression)	计算数值的自然对数
LOG10	float	LOG10(numeric_expression)	计算数值的常用对数
CEILING	整数类型	CEILING(numeric_expression)	计算大于或等于指定数值的最小整数
FLOOR	整数类型	FLOOR(numeric_expression)	计算小于或等于指定数值的最大整数
SIGN	tinyint	SIGN(numeric_expression)	计算数值的符号，正数返回1，零返回0，负数返回–1
RAND	float	RAND(seed)	返回一个0到1之间的随机小数
SIN	float	SIN(numeric_expression)	计算指定角度的正弦值
ASIN	float	ASIN(numeric_expression)	计算指定角度的反正弦值
COS	float	COS(numeric_expression)	计算指定角度的余弦值
ACOS	float	ACOS(numeric_expression)	计算指定角度的反余弦值
TAN	float	TAN(numeric_expression)	计算指定角度的正切值
ATAN	float	ATAN(numeric_expression)	计算指定角度的反正切值
COT	float	COT(numeric_expression)	计算指定角度的余切值
ACOT	float	ACOT(numeric_expression)	计算指定角度的反余切值

提示： 表 10-6 中的 numeric_expression 是任意可以产生数值类型数据的 T-SQL 表达式。

从表 10-6 中可以看出，T-SQL 的数学运算函数提供了完整的数学运算功能，而且使用非常简单，只要参数为数值类型的表达式即可。另外，不同的函数返回的数据类型也不一样，但是这些数学函数中除了 RAND()之外，全部都具有确定性，即任意时刻只要传入相同的参数进行运算，得到的计算结果肯定是一致的。

示例代码 10-18 演示数学函数的具体使用，定义 float 类型变量@num，先后设置它的值，并进行 ABS、CEILING、FLOOR、SIGN、SQRT 等运算。

示例代码 10-18

```
DECLARE @num float
SET @num = -1.4444;
PRINT 'Result 1...'
SELECT ABS(@num), CEILING(@num), FLOOR(@num), SIGN(@num);
SET @num = 2.0;
PRINT 'Result 2...'
SELECT SQRT(@num), SQUARE(@num), POWER(@num, 3), LOG(@num), LOG10(@num);
```

示例代码 10-18 的输出如下：

```
Result 1...
Column1       Column2       Column3       Column4
-------       -------       -------       -------
1.4444          -1            -2            -1
Result 2...
Column1       Column2       Column3       Column4       Column5
-------       -------       -------       -------       -------
1.41421         4             8           0.69314       0.30102
```

注意：示例代码 10-18 中的 PRINT 是 T-SQL 中用来打印输出信息的函数。而且 SELECT 语句也不是一定要查询数据库中的数据，也可以查询临时产生的数据。

10.3.7　使用 T-SQL 时间日期函数

时间日期是软件和数据库处理中常见的数据之一，T-SQL 也提供几个常用的日期和时间处理函数，包括获取当前时间、取得日期的年、取得日期的月等。表 10-7 列出了 T-SQL 中日期时间函数及其功能介绍。

表 10-7　T-SQL中常见的时间函数

函 数 名	返 回 类 型	格　　式	说　　明
GETDATE	Datetime	GETDATE()	获取系统的当前日期和时间
GETUTCDATE	Datetime	GETUTCDATE()	获取系统当前的UTC日期和时间
DATENAME	字符串	DATENAME(datepart, date)	获取指定日期时间的字符串格式，datepart表示要计算的时间部分
DATEADD	Datetime	DATEADD(datepart, number, date)	计算指定日期date加上一个时间间隔number之后的新时间，datepart表示要计算的时间部分
DATEDIFF	整数	DATEDIFF(datepart, startdate, enddate)	计算enddate减去startdate之后的值，datepart表示要计算的时间部分
DATEPART	整数	DATEPART(datepart, date)	计算指定日期date指定部分的整数，datepart表示要计算的时间部分
DAY	整数	DAY(date)	获取表示指定日期date的“日”的整数
MONTH	整数	MONTH(date)	获取表示指定日期date的“月”的整数
YEAR	整数	YEAR(date)	获取表示指定日期date的“年”的整数

提示：表 10-7 中的参数 datepart 表示要日期的哪一个部分，包括 year（年）、month（月）、day（日）、hour（时）、minute（分）、second（秒）、millisecond（毫秒）、

qiarter（季度）、dayofyear（当年的第几天）、week（当年的第几周）、weekday（周次）。

从表 10-7 中可以看出，T-SQL 的时间函数可执行几乎所有的日期和时间相关运算，但值得注意的是，它们只是计算日期时间的某一个部分，通过 datepart 参数来指定。如示例代码 10-19 所示，首先定义一个 datetime 类型的变量@curDate，然后通过 GETDATE()函数获取当前系统时间并赋值到@curDate。接下来的两个 SELECT 命令分别调用 YEAR()、MONTH()、DAY()等方法获取该时间的特定值。

示例代码 10-19

```
DECLARE @curDate datetime;
SET @curDate = GETDATE();
SELECT @curDate, YEAR(@curDate), MONTH(@curDate), DAY(@curDate);
SELECT DATENAME(hour,@curDate), DATENAME(minute, @curDate), DATENAME
(second, @curDate);
```

示例代码 10-19 产生的查询结果如下，可见当前时间为 2009-01-05 23:03:05。注意，由于计算的是系统当前时间，所以不同时刻运行，输出会不一样。

```
Column1         Column2         Column3         Column4
-------         -------         -------         ------
2009-01          2009              1               5
Column1         Column2         Column3
-------         --------        -------
23               3                 5
```

10.3.8　使用 T-SQL 字符串函数

在数据库中存储的大部分数据会以字符串形式保存，所以对字符串的处理在 T-SQL 中显得十分重要，T-SQL 提供了大量字符串处理函数，可以进行常见的字符和字符串运算。表 10-8 中列出了这些函数及其功能介绍。

表 10-8　T-SQL中常见的字符串函数

函　数　名	返 回 类 型	格　　式	说　　明
ASCII	整数	ASCII(character_expression)	获取字符串中第一个（最左边）字符的ASCII值
UNICODE	整数	UNICODE(character_expression)	获取字符串中第一个（最左边）字符的UNICODE值
CHAR	char	CHAR(integer_expression)	将一个整数转换成ASCII字符
NCHAR	nchar	NCHAR(integer_expression)	将一个整数转换成UNICODE字符
LEN	整数	LEN(character_expression)	获取字符串的字符数，不包括最后的连续空格

续表

函　数　名	返 回 类 型	格　　式	说　　明
LOWER	字符串	LOWER(character_expression)	将指定字符串转换为全部小写字母
UPPER	字符串	UPPER(character_expression)	将指定字符串转换为全部大写字母
STR	字符串	STR(float_expression, length, decimal)	将指定的数值转换为特定长度和精度的字符串，长度和精度可以不指定
SPACE	字符串	SPACE(integer_expression)	生成指定长度的全空格字符串
REPLICATE	字符串	REPLICATE(character_expression, integer_expression)	生成一个字符串，它包含指定重复次数的参数字符串
CHARINDEX	整数	CHARINDEX(expression1, expression2, start_location)	返回字符串expression1在字符串expression2中的索引，start_location表示开始查找的位置
PATNDEX	整数	PATINDEX(patern, character_expression)	返回正则表达式在字符串character_expression中的索引
LEFT	字符串	LEFT(character_expression, integer_expression)	返回字符串中从左边开始的指定长度的子字符串
RIGHT	字符串	RIGHT(character_expression, integer_expression)	返回字符串中从左边开始的指定长度的子字符串
LTRIM	字符串	LTRIM(character_expression)	去掉字符串左边的前导空格
RTRIM	字符串	RTRIM(character_expression)	去掉字符串右边的后导空格
REVERSE	字符串	REVERSE(character_expression)	返回字符串的逆序字符串
SOUNDEX	字符串	SOUNDEX(character_expression)	返回四个字符表示的代码，用于评估两个字符串的相似程度
DIFFERENCE	整数	DIFFERENCE(expression1, expression2)	返回一个取值为0～4整数，判断两个字符串的差异，0表示完全不一样，4表示几乎相同
REPLACE	字符串	REPLACE(expression1, expression2, expression3)	用expression3替换在expression1中出现的所有的expression2

注意：表 10-8 中的 expression 和 character_expression，都表示任何类型为字符串的 T-SQL 表达式，integer_expression 表示任何类型为整数的 T-SQL 表达式。

从表 10-8 中可以看出，T-SQL 的字符串处理函数功能非常全面，既可以计算字符串长度，可以进行去掉空格的运算，也可以查找和替换子串，甚至还支持正则表达式查找，比较两个字符串的相似性等。如示例代码 10-20 所示，首先声明一个字符串变量@str，然后设置它的值，最后将不同的字符串函数计算它们的值并输出。

示例代码 10-20

```
DECLARE @str varchar(25);
SET @str = 'This is a test string';
SELECT UPPER(@str) AS UpperCase, LEFT(@str, 4) AS FirstWord, LEN(@str) AS
Length;
SELECT ASCII(@str) AS AsciiCode, LEFT(@str, 1) AS Charactor;
SELECT REPLACE(@str, 'This', 'That');
```

示例代码 10-20 的输出如下，其中，所有的函数结果都和表 10-8 所介绍的一样。

```
UpperCase                FirstWord            Length
---------                ----------           --------
THIS IS A TEST STRING    This                 21
AsciiCode      Charactor
---------      ---------
84             T
Column1
---------------------------
That is a test string
```

在 T-SQL 中，函数常用在存储过程中，结合 T-SQL 语句和运算符可以开发出功能强大的数据处理程序，也可以大大提高数据查询和处理的效率。由于篇幅关系，本章的示例代码给出的所有函数示例都是简单的用法，读者有兴趣可以做更多试验。

10.4　使用 SQL Server 存储过程

在 SQL Server 中数据处理尤其是复杂的数据处理逻辑都可以通过存储过程来完成，存储过程是注册并运行在 SQL Server 服务器的 T-SQL 代码块，具有高效和安全的特点。本节介绍如何实现 SQL Server 存储过程。

10.4.1　存储过程介绍

在实际的软件开发过程中，经常会有一些比较常用或复杂的数据处理工作，用简单的 SQL 语句并不能解决问题，这就需要用到存储过程（Stored Procedure）。简单地说，存储过程是一组完成了特定功能的 SQL 指令集，在 SQL Server 2008 中，这些指令通常是 T-SQL 指令。存储过程在编译之后被注册并存储在 SQL Server 2008 数据库服务器中，用户通过 T-SQL 执行对应的存储过程即可，这样可以减少重复工作。

在微软的 SQL Server 2008 中，存储过程不仅仅是一对 SQL 命令而已，它与其他编程

语言（如 C#）中的过程类似，原因是存储过程可以：

- ❑ 存储过程可以接受输入参数并以输出参数的格式向调用者返回多个值。
- ❑ 存储过程可以在数据库中执行任何操作（包括调用其他存储过程）。
- ❑ 存储过程可以向调用者返回状态值，表明执行结果（如成功或失败、提示信息等）。

在 T-SQL 中，可以通过 EXECUTE 命令执行一个存储过程，但是存储过程与函数不同，存储过程不返回取代其名称的值，也不能直接在表达式中使用。在 SQL Server 2008 中存储过程具有以下优点：

- ❑ 存储过程已在服务器注册，所以更加安全、效率更高。
- ❑ 用户可以被授予权限来执行存储过程而不必直接对存储过程中引用的对象具有权限，还可以强制应用程序的安全性。
- ❑ 参数化存储过程有助于保护应用程序不受 SQL 注入的攻击。
- ❑ 存储过程允许模块化程序设计，可以完成更加复杂的数据处理逻辑。
- ❑ 存储过程创建以后，可以在程序中调用任意多次，使得应用程序更加易于维护，并且允许应用程序统一访问数据库。
- ❑ 存储过程可以减少网络通信流量。一个需要数百行 T-SQL 代码的操作可以通过一条执行过程代码的语句来执行，不需要在网络中发送数百行代码和大量数据。

在 SQL Server 2008 中，提供了 3 种类型的存储过程：系统存储过程、自定义存储过程和扩展存储过程。SQL Server 2008 中几乎所有的数据库管理活动都是通过系统存储过程来执行，系统存储过程存储在源数据库中，并且通常带有“sp_”前缀，例如，sp_changedbowner 就是一个系统存储过程。系统存储过程通常在成功执行之后返回 0，否则返回非 0。任何一个数据库都具有几乎相同的系统数据库，可以在 SQL Server Management Studio 的“可编程性”|“存储过程”|“系统存储过程”中找到所有的系统的存储过程，如图 10-9 所示。



图 10-9　系统存储过程例子

实际开发中最常用的是创建开发人员自己的存储过程——自定义存储过程，在 SQL Server 2008 中，开发人员可以创建两种存储过程：T-SQL 存储过程和 CLR 存储过程。T-SQL 存储过程是指用 T-SQL 命令开发的存储过程，它可以接收和返回用户提供的参数，也可以从数据库向客户端应用程序返回数据。CLR 存储过程是指对 Microsoft .NET Framework 公共语言运行时（CLR）方法的引用，可以接受和返回用户提供的参数，它们在.NET Framework 程序集中是作为类的公共静态方法实现。本章将重点介绍如何开发 T-SQL 存储过程。

注意：扩展存储过程在后续的 SQL Server 版本中将删除，取而代之的是 CLR 存储过程，本书不做进一步介绍，在实际的开发中也要尽可能避免使用这类存储过程。

10.4.2　创建 T-SQL 存储过程脚本

通常，可以通过两种方式来创建存储过程：通过 SQL Server Management Studio 创建存储过程，或者通过 Visual Studio 2010 的数据库项目创建存储过程。在 SQL Server Management Studio 中，需要如下几步骤创建和执行一个存储过程：

（1）打开 SQL Server Management Studio，并连接到要操作的数据库所在的数据库服务器。

（2）在“对象资源管理器”中选择需要创建存储过程的数据库，这里选择 UserLog2。

（3）打开数据的“可编程性”|“存储过程”结点，在右键快捷菜单中选择“新建存储过程”选项，如图 10-10 所示。



图 10-10　创建存储过程菜单

（4）SQL Server Management Studio 会自动产生一个存储过程的基本格式，编写存储过程的代码。

（5）通过右键快捷菜单或工具栏的“执行”命令执行该 SQL 脚本，就可以创建存储过程到指定的数据库，这里是 UserLog2。

在 Visual Studio 2010 中创建存储过程更加简单，它是基于数据库项目基础上执行的。首先打开 10.4.1 节创建的数据库项目 UserLogDbPrj，然后通过以下几个步骤创建一个存储过程。

（1）在“解决方案资源管理器”中右击 UserLogDbPrj 结点，选择“添加 SQL 脚本”命令，打开“创建新项”对话框。

（2）在“创建新项”对话框中选择“存储过程脚本”项目，在“名称”文本框中输入脚本名称，这里为 CreateUser，如图 10-11 所示。



图 10-11　“添加新项”对话框

（3）单击“添加”按钮，添加一个新的存储过程。

（4）编写存储过程脚本代码，通过右键菜单或工具栏的“执行”命令执行该 SQL 脚本，

就可以创建存储过程到指定的数据库。

注意：上面两种方法创建的存储过程完全没有区别，只是方式上不一样。但是后者，需要通过 USE 命令指定要操作的数据库。

10.4.3　编写简单 T-SQL 存储过程

从本质上讲，存储过程可以看成是一个函数，它具有名称、参数、代码块等。在 SQL Server 2008 中，存储过程的基本格式如下：

```
CREATE PROCEDURE <Procedure_Name>
    <@Param1 > <Datatype For Param1> = <Default Value For Param1>,
    <@Param2 > <Datatype For Param2> = <Default Value For Param2>
AS
BEGIN
    SELECT <@Param1 >, <@Param2 >
END
```

从上面的定义可以看出，一个存储过程通常包括以下几部分。

- ❑ CREATE PROCEDURE：该命令表示需要创建一个存储过程，其中 Procedure_Name 表示存储过程的名称，通常是一个具有实际意义，能够描述存储过程功能的名称。
- ❑ 存储过程的参数：一个存储过程可以没有任何参数，也可以包括多个参数，每个参数之间用逗号“，”分隔。如上面格式中给出了两个参数 Param1 和 Param2，参数用符号“@”标识。参数必须指定数据类型，比如 varchar(20)、int、float 等。还可以通过等号“=”在定义时指定参数的默认值。如“@Param1 int = 10”表示参数 Param1 类型为 int，默认值为 10。
- ❑ AS 关键字：AS 是存储过程名称和执行代码的分隔符。
- ❑ BEGIN…END：存储过程的代码块用关键字 BEGIN 开始，用 END 结束。这里的 BEGIN 和 END 类似于 C#语言里面的大括号“{ }”。至于代码块里面的内容，则由存储过程的功能决定。

说明：存储过程作为一个函数，通常需要一定的描述信息。简单描述这个存储过程的功能，创建时间和作者等，当然这些是注释信息，不是必需的。

10.4.4　安装和执行 T-SQL 存储过程

一个 T-SQL 存储过程可以完成任何功能，可以简单地查询数据库中的数据，也可以修改数据库中的数据，甚至可以执行数据库管理类操作，可以创建数据表、关系等。在前面介绍的 SQL 脚本中只是存储过程的代码，存储过程可用之前，需要安装存储过程到数据库。

简单的存储过程往往不包含任何参数，这种存储过程通常只包含一个名称，如示例代码 10-21 所示。创建了一个名为 UpdateUserAge 的存储过程，首先查询数据表 Users 中所有的用户及其年龄，然后通过 UPDATE 命令将年龄大于 25 的用户年龄加 1，最后重新查询 Users 中所有的用户及其年龄，比较前后数据的变化。

示例代码 10-21

```
IF EXISTS(SELECT*FROMsysobjectsWHERE type='P'ANDname='UpdateUserAge')
BEGIN
    DROP  Procedure  UpdateUserAge
END
GO

CREATE PROCEDURE UpdateUserAge
AS
BEGIN
    SELECT Users.LoginID, Users.Age
    FROM Users;

    UPDATE Users
    SET Users.Age = Users.Age+1
    WHERE Users.Age>25;

    SELECT Users.LoginID, Users.Age
    FROM Users;
END
GO
```

注意：在创建存储过程之前要保证新建的存储过程不存在，所以通常需要先判断是否已经存在存储过程，如果存在则删除。示例代码 10-21 中最前面的 IF 命令就是完成该功能。

执行示例代码 10-21 所示的 SQL 脚本，可以在数据表 UserLog2 中创建或更新存储过程 UpdateUserAge。然后在 SQL Server Management Studio 中，打开数据库 UserLog2 的“可编程性”|“存储过程”结点，可以看到存储过程 UpdateUserAge，如图 10-12 所示。

注意：如果在执行 SQL 脚本之前已经打开“可编程性”|“存储过程”结点，需要选择右键快捷菜单中的“刷新”命令才能显示新创建的存储过程。

创建并将存储过程安装到数据库之后，可以通过两种方式执行存储过程。第一种是通过 SQL Server Management Studio 的“执行存储过程”命令执行选中的存储过程，如图 10-13 所示。



图 10-12　UpdateUserAge 存储过程

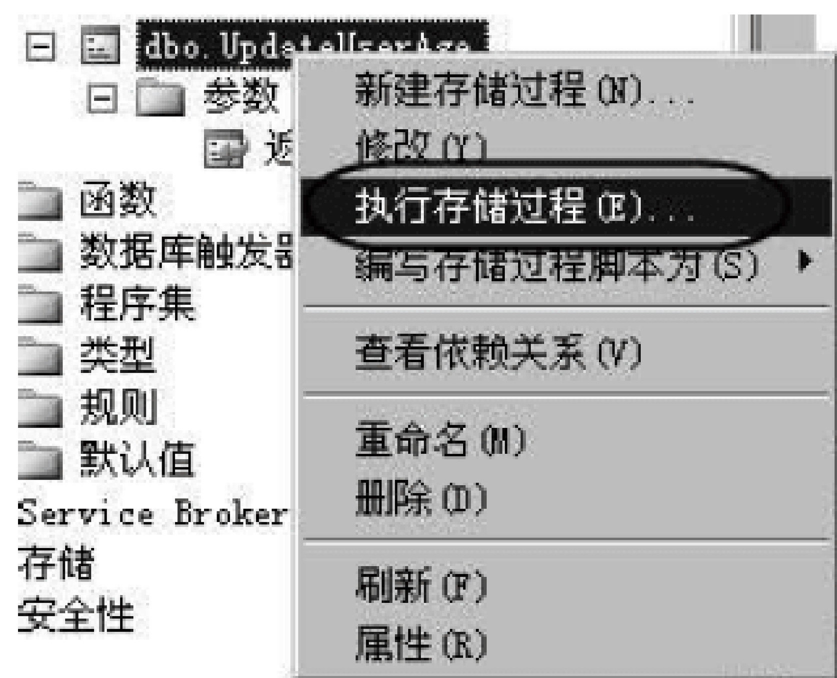


图 10-13　UpdateUserAge 存储过程

通过 SQL Server Management Studio 可以查看非常详细的存储过程执行结果，包括每一次查询产生的结果和存储过程返回的结果，如图 10-14 所示。另外，在“消息”页会给出存储过程在执行过程中产生的所有消息，包括自动产生的输出信息（比如 0 行数据受到

影响），也包括存储过程通过 PRINT 等命令打印的提示信息。

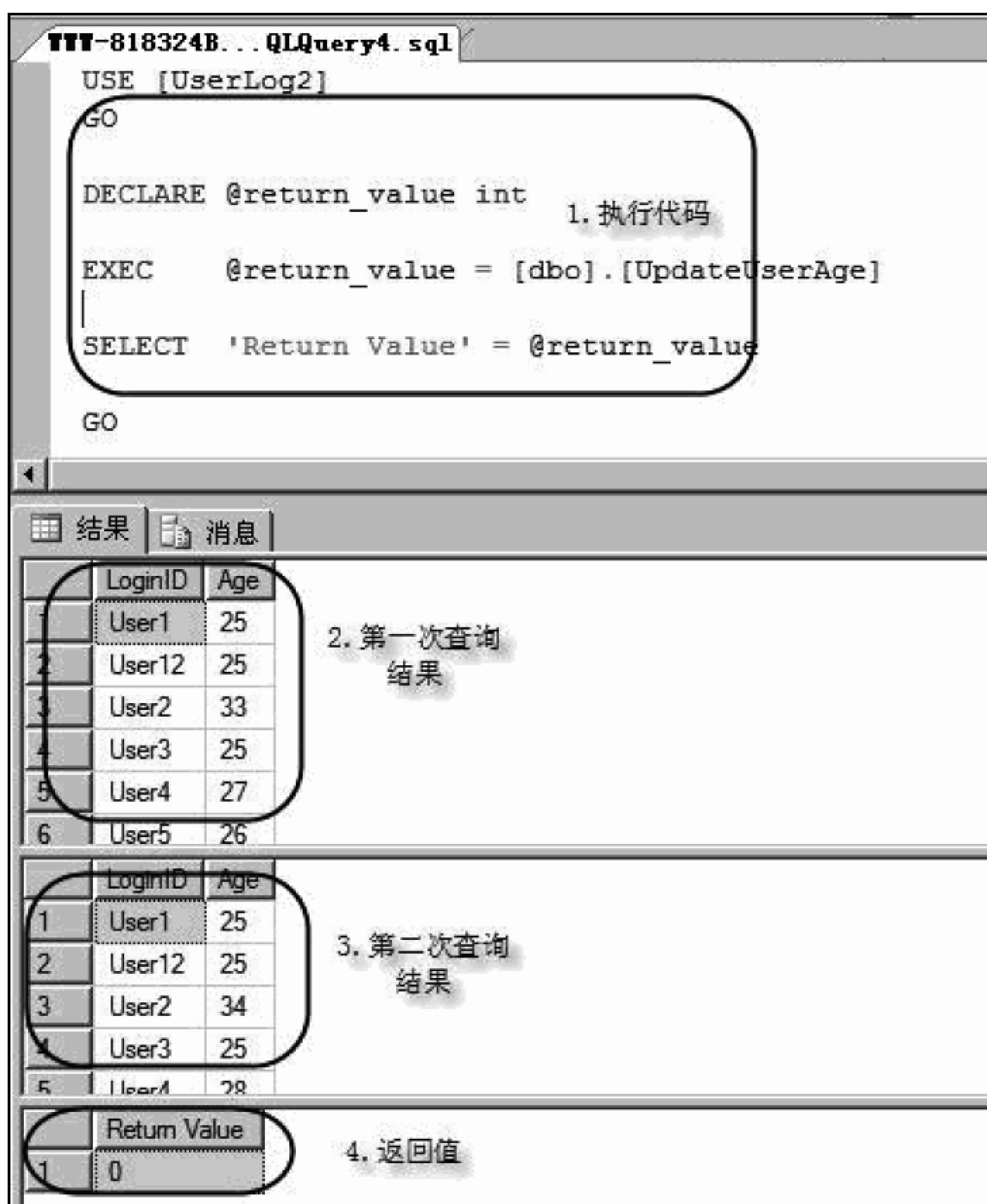


图 10-14　UpdateUserAge 存储过程执行结果

细心的读者已经从 10-14 中看出了第 2 种调用存储过程的方法，就是通过 T-SQL 的 EXEC 命令执行指定的存储过程。EXEC 命令需要指定存储过程的名称和需要的参数，如图 10-14 的第一部分所示。

注意：本质上讲第一种方法最终也是通过调用 EXEC 命令来执行存储过程，只是它提供了一种自动的执行代码生成和参数输入等功能，使用更加方便。

10.4.5　编写带参数的 T-SQL 存储

对于相对比较复杂的应用来说，通常需要存储过程具有参数和返回值。一个存储过程只能具有一个返回值，而且只能是 int 类型，返回值可以通过 RETURN 命令指定。但是一个存储过程可以包含一个或多个参数，每个参数都具有自己的数据类型（可以是任意数据类型），但参数名不能重复，也可以为参数指定默认值。存储过程包括两种类型的参数：输入参数和输出参数。

- 输入参数：没有 OUTPUT（或 OUT）关键字修饰的存储过程参数，只能为存储过程传入数据。

- 输出参数：用 OUTPUT（或 OUT）关键字修饰的存储过程参数，既可以为存储过程传入数据，还可以在存储过程中设置它们的值，外部调用者可以使用存储过程中对这些数据的修改。

示例代码 10-22 演示存储过程的参数和返回值的实例。其中存储过程 FindMaxMinAge 包括 3 个参数：int 类型的输入参数@rtType、int 类型的输出参数@maxAge、int 类型的输出参数@minAge。该存储过程查询数据表 Users 中所有用户的年龄最大值和最小值，分别保存在参数@maxAge 和@minAge 中，并且返回给调用者。其中参数@rtType 作为输入参数，如果为 1，则返回值为@maxAge，否则返回@minAge。

示例代码 10-22

```
IF EXISTS(SELECT * FROM sysobjects WHERE type='P' AND name='FindMax MinAge')
    BEGIN
        DROP  Procedure  FindMaxMinAge
    END
GO

CREATE Procedure FindMaxMinAge
    @rtType int ,
    @maxAge int OUTPUT,
    @minAge int OUTPUT
AS
BEGIN
    SET @maxAge =
        (SELECT MAX(Users.Age)
        FROM Users);
    SET @minAge =
        (SELECT MIN(Users.Age)
        FROM Users);
    IF @rtType = 1
        RETURN @maxAge;
    ELSE
        RETURN @minAge;
END
GO
```

示例代码 10-23 演示存储过程 FindMaxMinAge 的使用，执行结果输出如图 10-15 所示。第一次 EXEC 执行 FindMaxMinAge 产生的结果 Return Value 1 为@minAge 的值 25，因为参数@rtType 为 0。第二次 EXEC 执行 FindMaxMinAge 产生的结果 Return Value 2 为@maxAge 的值 34，因为参数@rtType 为 1。

示例代码 10-23

```
USE [UserLog2]
GO
DECLARE @return_value int,
            @maxAge int,
```

```
            @minAge int

EXEC    @return value=[dbo].[FindMaxMinAge]
        @rtType = 0,
        @maxAge = @maxAge OUTPUT,
        @minAge = @minAge OUTPUT

SELECT  @maxAge as N'@maxAge',
        @minAge as N'@minAge'
SELECT  'Return Value 1' = @return value

EXEC    @return_value = [dbo].[FindMaxMinAge]
        @rtType = 1,
        @maxAge = @maxAge OUTPUT,
        @minAge = @minAge OUTPUT
SELECT  'Return Value 2' = @return_value
```

结果　消息

	@maxAge	@minAge
1	34	25

	Return Value 1
1	25

	Return Value 2
1	34

图 10-15　FindMaxMinAge 存储过程执行结果

如果存储过程的参数为输入参数，而且没有指定默认值，那么在执行该存储过程时一定要设置该参数的值，否则会出现执行异常。同样的道理，如果 RETURN 命令返回的值或表达式不能转换为 int 类型时，也会产生执行异常。

技巧： 存储过程可以嵌套，也就是说存储过程可以调用其他的存储过程，但是存储过程的嵌套层次最多只能为 32 层，即它不支持递归运算。另外，存储过程本身还有比较高级的参数和使用方法，由于篇幅关系，本书并没有给出介绍。

10.5　小　　结

T-SQL 作为 SQL Server 2008 对 SQL 命令的扩展，具有非常强大的功能，可以执行任意的数据查询、更改和删除功能；也可以执行创建数据库、数据表、查询、关系等数据库管理操作；还可以执行用户管理、安全设置等数据库管理操作。

另外，T-SQL 还和其他的编程语言一样，支持多种数据类型的变量，带参数和返回值的函数，IF…ELSE 和 WHILE 跳转语句等，这些都使得 T-SQL 在数据库相关应用软件开发中变得十分有用。而存储过程作为 T-SQL 的一个重要特性，大大节省了重复的 T-SQL 代码，增加了重用性，减少了网络负载等。通过本章的学习读者应该掌握以下知识点：

- 什么是 T-SQL？T-SQL 能做什么？
- 如何在 Visual Studio 2010 中创建数据库项目？
- 如何通过 T-SQL 创建数据库和数据库表？
- 如何通过 T-SQL 创建数据表之间的关系？
- 如何插入记录到数据表？
- 如何更新和删除数据表中的记录？
- 如何查询数据表中的记录？如何进行多表查询和指定查询条件？
- T-SQL 支持哪些数据类型？
- T-SQL 支持哪些表达式？
- T-SQL 支持哪两个跳转语句？
- T-SQL 具有哪些内置函数？它们分别可以完成什么操作？
- 什么是 T-SQL 存储过程？它有什么优点？
- 如何创建 T-SQL 存储过程？
- 如何执行 T-SQL 存储过程？
- T-SQL 存储过程有几种类型的参数？

第 4 篇　ADO.NET 操作数据库

第 11 章　使用 ADO.NET 表示数据库

随着软件技术的发展，数据库技术在软件开发中变得更加广泛。另外，各大数据库软件开发商不断推出自己的管理软件，也使得数据库的统一访问变得日趋重要。面对多个相互并不兼容的数据库平台及各自的开发和编程接口，数据库相关开发需要一套通用而简单的接口。本章将介绍.NET 下统一的数据库访问组件——ADO.NET。

11.1　ADO.NET 简介

ADO.NET 是微软.NET 框架的一部分，它由一组工具和类库组成，应用程序通过它可以轻松地与基于文件或服务器的数据存储进行通信或管理。本节将介绍 ADO.NET 的基本概念。

11.1.1　什么是 ADO.NET

目前主流的数据库结构有很多种，既可以是基于文件系统的 Access、XML 文件等，也可以是基于服务器的数据库架构，比如 Oracle、Microsoft SQL Server、MySQL、DB2 等。同时这些数据库结构本身还有各自不同的版本，这一切都让统一的数据库访问接口变得日趋重要。

ADO.NET 是一组向.NET 程序员公开数据访问服务的接口（包括类、结构体、接口等）。ADO.NET 为创建分布式数据共享应用程序提供了一组丰富的组件，它对 Microsoft SQL Server 和 XML 等数据源及通过 OLE DB 和 XML 公开的数据源提供一致的访问。数据共享使用者应用程序可以使用 ADO.NET 连接到这些数据源，并检索、处理和更新所包含的数据。

ADO.NET 通过数据处理，将数据访问分解为多个可以单独使用或一前一后使用的不连续组件。ADO.NET 包含用于连接到数据库、执行命令和检索结果的.NET Framework 数据提供程序。可以直接处理检索到的结果，或将其放入 ADO.NET DataSet 对象，以便与来自多个源的数据进行远程处理的数据组合在一起，以特殊方式向用户公开。ADO.NET DataSet 对象也可以独立于.NET Framework 数据提供程序使用，以管理应用程序本地的数据或源自 XML 的数据。

ADO.NET 并不是 ADO 为了适应新的.NET 框架而改进得到的版本，它是微软进行全新设计的新产品。与 ADO 的早期版本和其他数据访问组件相比，ADO.NET 具有以下好处。

- 互操作性：ADO.NET 应用程序可以利用 XML 的灵活性和广泛性。由于 XML 是用于在网络中传输数据集的格式，因此可以读取 XML 格式的任何组件都可以处理数据。实际上，接收组件根本不必是 ADO.NET 组件，传输组件可以只是将数据集

传输给其目标，而不考虑接收组件的实现方式。目标组件可以是 Visual Studio 应用程序或无论用什么工具实现的其他任何应用程序，唯一的要求是接收组件能够读取 XML。

- 可编程性：.NET 类库中的 ADO.NET 数据组件以不同方式封装数据访问功能，帮助开发人员加快编程速度并减少犯错几率。已声明类型的数据集的代码更安全，原因在于它提供对类型的编译时检查。
- 性能：对于无链接的应用程序，ADO.NET 数据库提供的性能优于 ADO 无链接的记录集。
- 可伸缩性：ADO.NET 通过鼓励程序员节省有限资源来实现可缩放性，由于所有 ADO.NET 应用程序都使用对数据的无连接访问，因此它不会在较长持续时间内保留数据库锁或活动数据库连接。

由此可见，相比于 ADO 而言，ADO.NET 具有非常大的优势。本章将首先介绍 ADO.NET 中比较重要的两个类：数据集（DataSet）和数据表（DataTable）。

11.1.2　ADO.NET 数据提供程序

在 ADO.NET 中，对数据库的操作被分成连接到数据库、执行数据库命令、返回命令执行结果 3 个可以独立进行的步骤，而开发人员真正需要处理的是数据到达内存中 DataSet 对象之后的操作。数据提供程序就是执行数据库相关操作的核心组件，如图 11-1 所示，不同的数据库提供程序支持不同类型的数据库，开发人员可以根据需要开发特定的数据库实现数据提供程序。

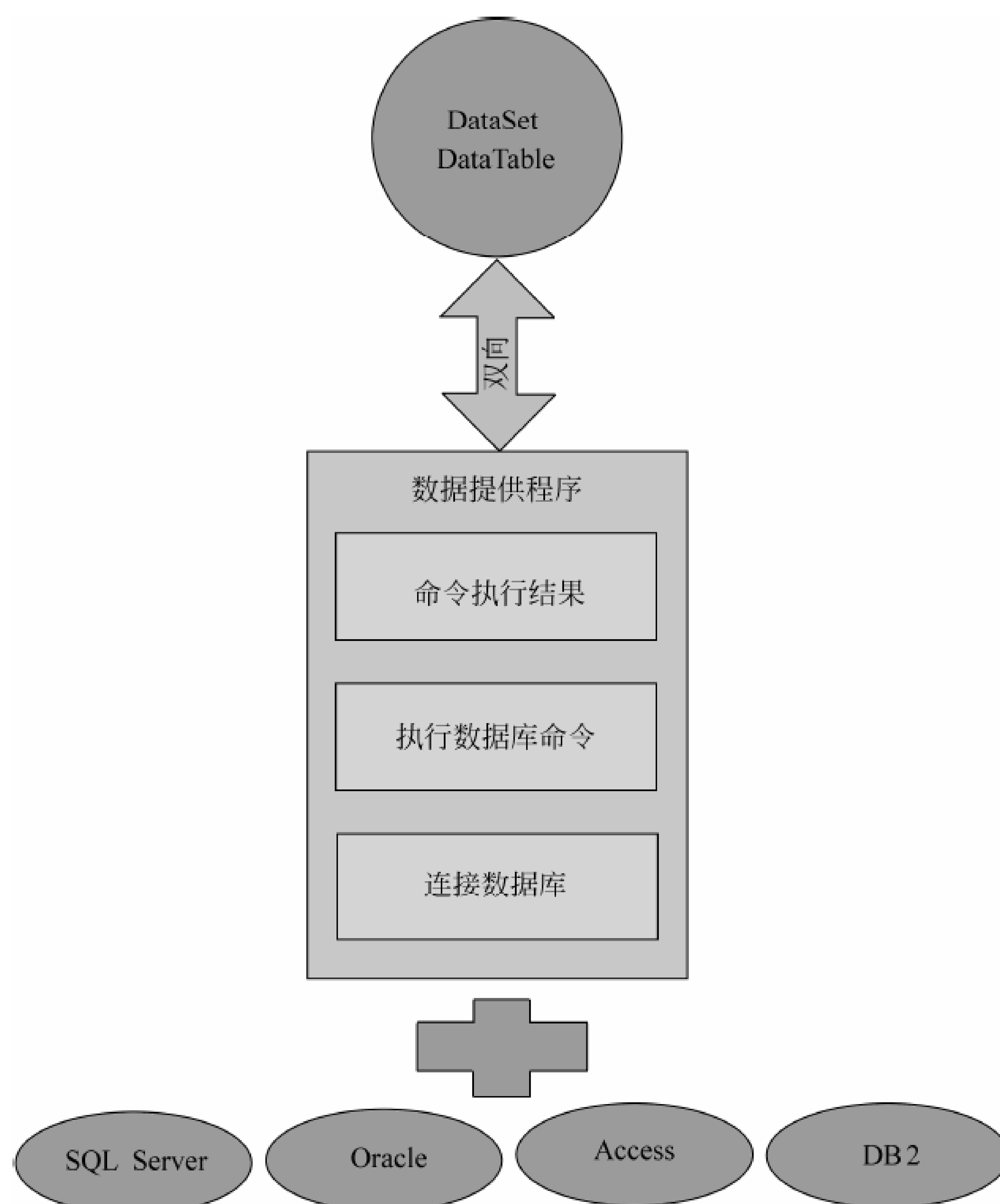


图 11-1　ADO.NET 数据提供程序

在 ADO.NET 中，.NET 类库对目前最流行的数据库都提供了各自的数据提供程序，也就是说目前 ADO.NET 可以支持当前流行的各种数据库的数据访问。不同数据库的数据提供程序封装在不同的.NET 类库中，如表 11-1 所示，ADO.NET 内置的数据提供程序可以支持 SQL Server、ODBC、Access、Oracle 等。

表 11-1　ADO.NET内置的数据提供程序

ADO.NET 数据提供程序	命 令 空 间	说　　明
.NET Framework Data Provider for SQL Server	System.Data.SqlClient	提供对Microsoft SQL Server 7.0或更高版本中数据的访问
.NET Framework Data Provider for OLE DB	System.Data.OleDb	提供对使用 OLE DB 公开的数据源中数据的访问。如 SQL Server、Access 等
.NET Framework Data Provider for ODBC	System.Data.Odbc	提供对使用 ODBC 公开的数据源中数据的访问
.NET Framework Data Provider for Oracle	System.Data.OracleClient	适用于Oracle数据源。用于Oracle的.NET Framework 数据提供程序支持 Oracle 客户端软件 8.1.7 和更高版本

注意：在使用这些内置的数据提供程序，访问对应数据库服务器中的数据之前，首先要通过 Using 关键字导入对应命名空间。

11.1.3　了解 ADO.NET 相关类库

在 ADO.NET 数据提供程序中，包括多个核心类，这些类抽象了 ADO.NET 中数据库访问各独立操作所需要实现的功能接口。如表 11-2 所示，其中每个核心类都表示一个独立的功能抽象，如果实现新的数据提供程序，就需要至少实现这些核心类。每个核心类都具有一个唯一的基类，而且这些基类都以 Db 为前缀进行命名。

表 11-2　ADO.NET的核心类

核　心　类	基　　类	说　　明
数据库连接（Connection）	DbConnection	建立并表示与数据库服务器的连接
数据库命令（Command）	DbCommand	表示并执行特定的数据库命令
数据读取器（DataReader）	DbDataReader	表示从数据库服务器以只读向前的方式获取数据的数据流
数据适配器（DataAdapter）	DbDataAdapter	使用数据库服务器中的数据填充DataSet或将DataSet的更改更新到数据库服务器
事务（Transaction）	DbTransaction	在数据库服务器登记事务
命令生成器（CommandBuilder）	DbCommandBuilder	自动为DataAdapter生成需要执行的数据库命令，并指定命令的参数等
连接字符串生成器（ConnectionStringBuilder）	DbConnectionStringBuilder	自动产生与Connection对象相对应的数据库连接字符串文本
参数（Parameter）	DbParameter	定义数据库命令的输入、输出、返回值等参数信息

🔔技巧：在进行数据库相关的接口定义时，可以尽可能地使用基类而不是使用特定的数据提供程序的类，这样可以使得接口更加灵活和通用。

11.2　DataTable 数据表

类 DataTable 是 ADO.NET 中主要的类之一，是一个数据表在内存中的表现形式。通过对 DataTable 对象的处理可以在内存中进行数据修改、回滚等操作，本节将介绍 DataTable 类的具体使用。

11.2.1　了解 DataTable 类成员

DataTable 类是一个数据表在内存中的表示形式，它包括数据表的列定义，DataColumnCollection 属性表示数据表中的列定义，通过数据提供程序可以自动从数据库服务器获取该信息，也可以通过代码的形式创建该信息。当以代码的方式创建 DataTable 时，先创建必须的 DataColumn 对象，然后将它们添加到 DataColumnCollection 中。只有获得了 DataTable 的列定义后才能添加它的数据记录。

通常，DataRow 表示 DataTable 中的一条记录，通过 DataTable 的 NewRow()方法可以获得一个满足 DataTable 列定义的 DataRow 对象，然后再设置新的 DataRow 的数据。一个 DataTable 最多可存储 16777216 条数据记录。

另外，DataTable 类也包含用于确保数据完整性的 Constraint 对象的集合，该集合可以指定多个数据表之间的关系。最后，有许多 DataTable 相关事件可用于确定数据记录发生更改，包括 RowChanged、RowChanging、RowDeleting 和 RowDeleted。表 11-3 列出了 DataTable 类的常用成员。

表 11-3　DataTable类常用成员

分类	名　　称	访问性	说　　明
属性	TableName	Public	读写属性，获取或设置DataTable的名称
	IsInitialized	Public	只读属性，指示是否已初始化DataTable
	HasErrors	Public	只读属性，读取表所属DataSet的任何表的任何行中是否有错误
	CaseSensitive	Public	读写属性，指示表中的字符串比较是否区分大小写
	ExtendedProperties	Public	只读属性，获取自定义用户信息的集合
	MinimumCapacity	Public	读写属性，获取或设置该表最初的起始大小、可容纳的记录数
	Namespace	Public	读写属性，获取或设置DataTable中所存储数据的XML表示形式的命名空间
	DataSet	Public	只读属性，获取此表所属的DataSet
	Columns	Public	只读属性，获取属于该表的列的集合
	PrimaryKey	Public	读写属性，获取或设置充当数据表主键的列的数组
	Rows	Public	只读属性，获取属于该表的行的集合
	DefaultView	Public	只读属性，获取可能包括筛选视图或游标位置的表的自定义视图

续表

分类	名　　称	访问性	说　　明
属性	DisplayExpression	Public	读写属性，获取或设置一个表达式，该表达式返回的值用于表示用户界面中的此表
	ChildRelations	Public	只读属性，获取此DataTable的子关系的集合
	ParentRelations	Public	只读属性，获取该 DataTable 的父关系的集合
	Constraints	Public	只读属性，获取由该表维护的约束的集合
方法	DataTable	Public	构造函数，创建DataTable数据表对象
	BeginInit	Public	开始初始化在窗体上使用或由另一个组件使用的DataTable
	EndInit	Public	结束在窗体上使用或由另一个组件使用的DataTable的初始化
	BeginLoadData	Public	在加载数据时关闭通知、索引维护和约束
	EndLoadData	Public	在加载数据后打开通知、索引维护和约束
	Load	Public	通过所提供的IDataReader，用某个数据源的值填充DataTable，如果 DataTable已经包含行，则从数据源传入的数据将与现有的行合并
	LoadDataRow	Public	查找和更新特定行，如果找不到任何匹配行，则使用给定值创建新行
	ImportRow	Public	将DataRow复制到DataTable中，保留任何属性设置及初始值和当前值
	Merge	Public	将指定的DataTable与当前的DataTable合并
	Copy	Public	复制该DataTable的结构和数据
	Clear	Public	清除DataTable中的所有记录数据
	Reset	Public	将DataTable重置为其初始状态
	GetErrors	Public	获取包含错误的DataRow对象的数组
	GetChanges	Public	获取DataTable的更改副本，它包含目前DataTable中所有未接受或拒绝的数据记录更改
	AcceptChanges	Public	提交目前DataTable中所有未接受或拒绝的数据记录更改
	RejectChanges	Public	回滚目前DataTable中所有未接受或拒绝的数据记录更改
	CreateDataReader	Public	返回与此DataTable中的数据相对应的DataTableReader
	NewRow	Public	创建与该表具有相同架构的新DataRow
	Select	Public	获取DataRow对象的数组
	ReadXml	Public	将XML架构和数据从XML数据流读入到DataTable
	WriteXml	Public	将DataTable的当前内容以XML格式写入到XML数据流
	ReadXmlSchema	Public	将XML架构从XML数据流读入到DataTable
	WriteXmlSchema	Public	将DataTable的当前数据结构以XML架构形式写入到XML数据流
事件	Initialized	Public	初始化DataTable后引发该事件
	TableNewRow	Public	插入新DataRow时引发该事件
	ColumnChanging	Public	在DataRow中指定的DataColumn的值发生更改时引发该事件
	ColumnChanged	Public	在DataRow中指定的DataColumn的值被更改后引发该事件
	RowChanging	Public	在DataRow正在更改时引发该事件
	RowChanged	Public	在成功更改DataRow之后引发该事件
	RowDeleting	Public	在表中的行要被删除之前引发该事件
	RowDeleted	Public	在表中的行已被删除后引发该事件
	TableClearing	Public	在清除DataTable时引发该事件
	TableCleared	Public	在清除DataTable后引发该事件

从表 11-3 中可以看出，DataTable 既可以通过 DataRow 添加数据，也可以从 XML 数据流中读取数据，它同样可以将数据写到 XML 文件。

11.2.2　添加和删除 DataTable 的记录

DataTable 类表示一个数据表，一般可以通过两种方式创建：

- 一是通过数据库操作从数据库获取；
- 二是直接通过 DataTable 的构造函数创建。

第一种方法将在后面章节详细介绍，本节主要介绍第二种方法。通过 DataTable 的构造函数创建一个数据表，通常需要下面 3 个步骤：

（1）通过 DataTable 类的构造函数创建一个 DataTable 对象。DataTable 类的构造函数具有两个常用版本，定义如下，通常需要为数据表指定一个合适的有实际意义的表名。

- DataTable()：创建一个表名为空字符串的数据表。
- DataTable(string name)：创建一个表名为 name 指定值的数据表。

（2）根据 DataTable 中列定义信息创建多个 DataColumn 对象，并依次添加到 DataTable 的 Columns 属性中。DataColumn 对象通常是直接通过它的构造函数来创建，常用的有以下 3 个版本。

- DataColunm(string name, Type ty)：创建一个列名为 name、类型为 ty 的数据列，DataType 可以是数据库支持的任意数据类型。
- DataColumn(string name, Type ty, string expr)：创建一个列名为 name、类型为 ty 的数据列。参数 expr 指定用于创建该列的表达式。
- DataColumn(string name, Type ty, string expr, MappingType mty)：创建一个列名为 name、类型为 ty 的数据列。参数 expr 指定用于创建该列的表达式，参数 mty 表示该列映射到 XML 数据源的结点类型。

（3）通过 DataTable.NewRow()方法获取符合当前表结构的 DataRow 对象，并为它设置对应字段的数据，然后将该 DataRow 对象添加到 DataTable 类的 Rows 属性中。

注意：对于单个 DataTable 对象，通常不需要为它指定关系信息（Constraints 属性），如果是多个有依赖关系的 DataTable 对象，则需要为它们都指定对应的关系信息。

如示例代码 11-1 中，方法 CreateUserTable()通过 DataTable 构造函数创建一个名为“Users”的数据表 dt。通过 DataColumn 的构造函数创建一个列 UserName，并添加到 dt 中，然后通过 DataColumnCollection.AddRange()一次性添加两个列 Age 和 Mobile 到 dt 中。接下来通过 DataTable.Rows 属性的 Add()方法添加两条记录到 dt 中。

方法 DisplayTableInfo()通过 foreach 语句遍历 DataTable.Columns 属性，从而得到数据表中所有列的信息，并打印出列信息，同时打印出数据表的其他常用信息。

示例代码 11-1

```
static void Main(string[] args)
{
    DataTable dt = CreateUserTable();
    DisplayTableInfo(dt);
```

```
}
public static DataTable CreateUserTable( )
{
    DataTable dt = new DataTable("Users");      //创建一个 DataTable 对象 dt
    //创建数据表 UserName 的列信息
    DataColumn col = new DataColumn("UserName", typeof(string));
    col.Caption = "姓名";                       //设置列的别名
    col.AllowDBNull = false;                    //设置列不可以为空
    dt.Columns.Add(col);                        //添加列岛数据表

    dt.Columns.AddRange(                        //再增加两个列信息
        new DataColumn[] {
            new DataColumn("Age", typeof(int)), //Age 列
            new DataColumn("Mobile", typeof(string)),    //Mobile 列
        });

    DataRow row = dt.NewRow( );                 //创建一个新记录，并修改记录的值
    row["UserName"] = "张三";
    row["Age"] = 20;
    row["Mobile"] = "13511112222";
    dt.Rows.Add(row);                           //添加记录 row 到 dt

    dt.Rows.Add(new object[] {                  //直接添加记录到 dt
        "李四", 22, "13112345678" });
    return dt;                                  //返回数据表
}
//显示数据表的信息
public static void DisplayTableInfo(DataTable dt)
{
    //显示数据表的列信息
    System.Console.WriteLine("数据表[{0}]的列信息:", dt.TableName);
    foreach (DataColumn col in dt.Columns)
    {
        System.Console.WriteLine("{0}\t{1}\t{2}\t{3}",
            col.ColumnName, col.Caption, col.DataType, col.AllowDBNull);
    }
    System.Console.WriteLine( );
    //显示数据表的名称、行数、是否区分大小写等信息
    System.Console.WriteLine("数据表[{0}]的其他信息: ", dt.TableName);
    System.Console.WriteLine("\t 行数:{0}", dt.Rows.Count);
    System.Console.WriteLine("\t 是否区分大小写: {0}", dt.CaseSensitive);
    System.Console.WriteLine("\t 是否具有错误:{0}", dt.HasErrors);
}
```

示例代码 11-1 的输出如下，从中可以看出，如果列信息没有指定别名，那么它的别名默认和列名相同。默认情况下列不能为空，在数据表中进行比较不区分大小写。

```
数据表[Users]的列信息:
UserName        姓名    System.String   false
Age     Age     System.Int32    true
Mobile  Mobile  System.String   true

数据表[Users]的其他信息:
        行数:2
        是否区分大小写: false
        是否具有错误:false
```

注意：一般地，通过代码方式创建的 DataTable 的列定义为静态的，这样就可通过 DataTable.Columns.AddRange()方法一次性添加多条记录，从而简化代码的编写。

11.2.3　遍历 DataTable 的记录

在 DataTable 中，所有记录都保存在 Rows 属性中，该属性是一个 DataRowCollection 类型，表示一组数据记录的集合，每个元素都是一个 DataRow 对象。通过 foreach 运算符遍历 DataTable.Rows 属性可以遍历 DataTable 中的所有记录。

DataRow 类型具有多个独立的索引器，通过这些索引器可以获取记录中指定列的数据，主要包括以下 3 个常用的版本。

- public object this[string columnName]：通过列名获取记录中指定列的数据。
- public object this[int columnIndex]：通过列索引（从 0 开始计数）获取记录中指定列的数据。
- public object this[DataColumn column]：通过表示数据列的 DataColumn 对象获取记录中指定列的数据。

所以，遍历 DataTable 中的数据通常分成两步，首先，通过 foreach 命令遍历它的 Rows 属性，然后，对每个 DataRow 对象进行处理，比如通过索引器来获取记录各字段的值。

示例代码 11-2 演示如何遍历 DataTable 中的所有记录，其中，PrintDataTableByName()方法直接通过列名获取记录中的数据，并打印数据。PrintDataTableByColumn()方法通过遍历 Columns 属性获取所有列，并通过 DataColumn 对象获取记录中的数据。

示例代码 11-2

```
static void Main(string[] args)
{
    DataTable dt = CreateUserTable();                   //创建数据表
    PrintDataTableByName(dt);                           //打印数据表
    PrintDataTableByColumn(dt);                         //打印数据表
}

public static DataTable CreateUserTable( )
{
    //此处省略创建用户表的代码，详情请看示例代码 11-1
}

public static void PrintDataTableByName(DataTable dt)
{
    System.Console.WriteLine("PrintDataTableByName():");
```

```
    foreach (DataRow row in dt.Rows)                    //遍历所有的记录
    {
        System.Console.Write("{0}\t", row["UserName"]);
                                                        //根据列名获取记录的数据
        System.Console.Write("{0}\t", row["Age"]);
        System.Console.Write("{0}\t", row["Mobile"]);
        System.Console.WriteLine( );
    }
}

public static void PrintDataTableByColumn(DataTable dt)
{
    System.Console.WriteLine("PrintDataTableByColumn():");
    foreach (DataRow row in dt.Rows)                    //遍历所有的记录
    {
        foreach (DataColumn col in dt.Columns)          //遍历数据表中所有的列
        {
                    System.Console.Write("{0}\t", row[col]);
                                                        //根据DataColumn获取记录
                                                        的数据
        }
        System.Console.WriteLine( );
    }
}
```

示例代码 11-2 的输出如下，两个方法都打印出数据表中的记录，从中可以看出数据表中包含两条记录，每条记录包括 3 列数据。

```
PrintDataTableByName():
张三    20      13511112222
李四    22      13112345678
PrintDataTableByColumn():
张三    20      13511112222
李四    22      13112345678
```

注意：通过列名或列索引获取记录的数据具有更大的局限性，完全依赖于 DataTable 中列的定义，如果列名或顺序发生变化，执行结果将发生变化。通过 DataColumn 对象获取记录的数据更加灵活，可以适用于任何结构的数据表。

11.2.4　提交或回滚 DataTable 的更改

在 ADO.NET 中，DataTable 本身可以脱离数据库进行使用，对 DataTable 数据的更改与数据库并没有关系，这些更改都是在内存中修改数据。DataTable 的使用者，可以通过 DataTable 的成员方法或属性来判断数据更改是否有错误，可以提交 DataTable 中所有数据的更改，也可以回滚（拒绝）数据的更改。

在 DataTable 中，通过 AccepteChanges()方法提交该 DataTable 在内存中的所有更改，通过 RejectChanges()方法拒绝该 DataTable 在内存中的所有更改，这两个函数定义如下。

- public void AcceptChanges()：提交自上次调用 AcceptChanges()以来对该表进行的所有更改。
- public void RejectChanges()：回滚自该表加载以来或上次调用 AcceptChanges()以来，对该表进行的所有更改。

示例代码 11-3 演示 AccepteChanges()和 RejectChanges()方法的使用，首先创建一个 DataTable 对象 dt，然后向 dt 中添加记录，并通过 AccepteChanges()方法提交更改，或者通过 RejectChanges()方法回滚更改，并打印出最新的记录数。

示例代码 11-3

```
public static void AccepteDataTable( )
{
    DataTable dt = new DataTable("Users");            //创建一个 DataTable
                                                       对象 dt
    dt.Columns.AddRange(                               //创建 UserName 列信息
        new DataColumn[] {
            new DataColumn("UserName", typeof(string)), //UserName 列
            new DataColumn("Age", typeof(int)),         //Age 列
            new DataColumn("Mobile", typeof(string)),   //Mobile 列
        });
    System.Console.WriteLine("1.DataTable 有{0}行数据.", dt.Rows.Count);

    dt.Rows.Add(new object[] {                         //添加 1 条记录到 dt
        "李四", 22, "13112345678" });
    System.Console.WriteLine("2.DataTable 有{0}行数据.", dt.Rows.Count);
    dt.RejectChanges( );                               //回滚新增的 1 条记录
    System.Console.WriteLine("3.DataTable 有{0}行数据.", dt.Rows.Count);

    dt.Rows.Add(new object[] {                         //添加 1 条记录到 dt
        "李四", 22, "13112345678" });
    dt.Rows.Add(new object[] {                         //添加 1 条记录到 dt
        "王五", 24, "13112345688" });
    dt.AcceptChanges( );                               //接受新增的 2 条记录
    System.Console.WriteLine("4.DataTable 有{0}行数据.", dt.Rows.Count);

    dt.Rows.Add(new object[] {                         //添加 1 条记录到 dt
        "岳飞", 44, "13112388888" });
    dt.RejectChanges( );                               //回滚新增的 1 条记录
    System.Console.WriteLine("5.DataTable 有{0}行数据.", dt.Rows.Count);
}
```

示例代码 11-3 的输出如下，从中可以看出，当 DataTable 新创建时只有 0 行数据（编号 1），在新增 1 条记录之后，DataTable 中有 1 条没有没有提交的记录（编号 2），然后通过 RejectChanges()方法回滚了新增的纪录，即取消新增操作，则记录数重新变为 0（编号 3）。同样地，又添加 2 两条新记录到 DataTable，然后通过 AccepteChanges()方法提交新增记录，此时记录数为 2（编号 4）。

```
1.DataTable 有 0 行数据.
2.DataTable 有 1 行数据.
3.DataTable 有 0 行数据.
4.DataTable 有 2 行数据.
5.DataTable 有 2 行数据.
```

在 DataTable 中，任何内存数据的更改，只要调用 AccepteChanges()提交更改之后，DataTable 不再包含任何更改，再调用 RejectChanges()方法不会影响到已经提交的记录。另外，AccepteChanges()和 RejectChanges()都可以作用于任何对 DataTable 记录的更改，包括新增记录、删除记录、修改记录等。

注意：在接受 DataTable 的更改之前，通常需要先通过 bool 类型的 DataTable.HasError 属性判断更改中是否存在错误，如果没有错误才提交更改，否则给出对应的处理，比如抛出异常或通知更高层的应用程序。

11.2.5　监视 DataTable 记录的更改

从表 11-3 中可以看出，DataTable 类提供了多个事件用来提醒 DataTable 的使用者当前正在发生的事情，所以可以通过响应这些事件监视 DataTable 的初始化、记录更改等更改。

DataTable 在初始化时会引发 Initialized 事件，通过响应该事件可以监视 DataTable 是否成功创建并初始化。DataTable 包含 5 个和数据记录有关的事件，分别在新建记录、删除记录、增加记录时产生，定义如下。

- TableNewRow：在调用 DataTable.NewRow()方法插入新记录时引发该事件，但是通过 Rows.Add()方法插入新记录时不会引发该事件。事件参数中包括新插入的 DataRow 对象。
- RowChanging 和 RowChanged：当 DataTable 中的记录发生更改之前引发 RowChanging 事件，更改完成后引发 RowChanged 事件。事件参数中包括发生更改的 DataRow 对象，同时还包括具体发生怎样的更改，包括添加、提交、回滚、删除等。
- RowDeleting 和 RowDeleted：当 DataTable 中的记录删除之前引发 RowDeleting 事件，删除完成后引发 RowDeleted 事件。事件参数包括被删除的 DataRow 对象。

另外，DataTable 还包括列定义相关的事件：ColumnChanging 和 ColumnChanged 事件，当 DataTable 的列定义发生更改时会依次引发这两个事件，事件参数包括被更改的 DataColumn 对象及对应的操作。为了方便全部删除数据，DataTable 还提供了 TableClearing 和 TableCleared 事件，这两个事件在调用 DataTable.Clear()方法时引发。

示例代码 11-4 演示如何使用这些事件，首先是创建 DataTable 对象 dt，然后绑定对应事件到特定的事件处理函数，在事件处理函数中可以做任何和业务逻辑相关的事情，作为示例代码，这里只是简单地提示当前发生的事件。

示例代码 11-4

```
public static void ListenDataTable( )
{
    DataTable dt = new DataTable("Users");    //创建一个 DataTable 对象 dt
    //绑定所有的事件处理函数
    dt.TableNewRow += new DataTableNewRowEventHandler(dt_TableNewRow);
    dt.TableCleared += new DataTableClearEventHandler(dt_TableCleared);
    dt.TableClearing += new DataTableClearEventHandler(dt_TableClearing);
    dt.RowChanged += new DataRowChangeEventHandler(dt_RowChanged);
```

```
    dt.RowChanging += new DataRowChangeEventHandler(dt RowChanging);
    dt.RowDeleted += new DataRowChangeEventHandler(dt RowDeleted);
    dt.RowDeleting += new DataRowChangeEventHandler(dt RowDeleting);
    dt.ColumnChanging += new DataColumnChangeEventHandler(dt ColumnChanging);
    dt.ColumnChanged += new DataColumnChangeEventHandler
    (dt ColumnChanged);
    dt.Columns.AddRange(        //创建 UserName 列，引发 ColumnChanging 和
                                ColumnChanged 事件
        new DataColumn[] {
            new DataColumn("UserName", typeof(string)),
        });
    DataRow row = dt.NewRow( ); //新建 Row，引发 TableNewRow 事件
    row["UserName"] = "李四";
    dt.Rows.Add(row);           //添加 Row，引发 RowChanging 和 RowChanged 事件
    dt.Rows.Add("王五");        //添加 Row，引发 RowChanging 和 RowChanged 事件
    dt.Rows.RemoveAt(0);        //删除 Row，引发 RowDeleting 和 RowDeleted 事件
    dt.Clear( );                //清空 DataTable，引发 TableClearing 和
                                TableCleared 事件
}

static void dt ColumnChanged(object sender, DataColumnChangeEventArgs e)
{
    System.Console.WriteLine("ColumnChanged...");      //打印提示信息
}
static void dt ColumnChanging(object sender, DataColumnChangeEventArgs e)
{
    System.Console.WriteLine("ColumnChanging...");     //打印提示信息
}
static void dt RowDeleting(object sender, DataRowChangeEventArgs e)
{
    System.Console.WriteLine("RowDeleting...");        //打印提示信息
}
static void dt_RowDeleted(object sender, DataRowChangeEventArgs e)
{
    System.Console.WriteLine("RowDeleted...");         //打印提示信息
}
static void dt RowChanging(object sender, DataRowChangeEventArgs e)
{
    System.Console.WriteLine("RowChanging...");        //打印提示信息
}
static void dt_RowChanged(object sender, DataRowChangeEventArgs e)
{
    System.Console.WriteLine("RowChanged...");         //打印提示信息
}
static void dt TableClearing(object sender, DataTableClearEventArgs e)
{
    System.Console.WriteLine("TableClearing...");      //打印提示信息
}
static void dt_TableCleared(object sender, DataTableClearEventArgs e)
{
    System.Console.WriteLine("TableCleared...");       //打印提示信息
}
static void dt TableNewRow(object sender, DataTableNewRowEventArgs e)
{
    System.Console.WriteLine("TableNewRow...");        //打印提示信息
}
```

示例代码 11-4 的输出如下，从中可以看出，ColumnChanging 和 ColumnChanged、

RowChanging 和 RowChanged、RowDeleting 和 RowDeleted、TableClearing 和 TableCleared 都是成对引发的，ing 形式的事件用于操作执行之前进行相关处理，ed 形式的事件用于操作执行之后进行相关处理。

```
TableNewRow...
ColumnChanging...
ColumnChanged...
RowChanging...
RowChanged...
RowChanging...
RowChanged...
RowDeleting...
RowDeleted...
TableClearing...
TableCleared...
```

11.3　DataSet 数据集合

在 ADO.NET 中，除了 DataTable 外，DataSet 也用于在内存中保存数据，DataSet 就好比一个内存中的小型数据库，它记录了所有关系型数据库正常工作所需要的信息。本节将介绍 DataSet 类的具体使用。

11.3.1　了解 DataSet 类成员

DataSet 类是 ADO.NET 的另外一个核心类，DataSet 类用来表示内存中的数据集合，可以将它看成是一个简单的内存数据库。一个 DataSet 可以包含多个 DataTable，并且可以包含 DataTable 之间的关系、限制等信息。另外，DataSet 同样可以表示 XML 数据，可以从 XML 数据流读取数据到 DataSet，也可以将 DataSet 中的数据写入到 XML 数据流。

DataSet 中所有的数据表都可以通过它的 Tables 属性访问，DataSet 中数据的存储实际是通过 DataTable 来实现。表 11-4 给出了 DataSet 类的常用成员。

表 11-4　DataSet类的常用成员

分类	名　　称	访问性	说　　明
属性	DataSetName	Public	读写属性，表示当前DataSet的名称
	Tables	Public	只读属性，表示包含在DataSet中的表（DataTable）的集合
	CaseSensitive	Public	读写属性，表示DataTable对象中的字符串比较是否区分大小写
	HasErrors	Public	只读属性，表示在DataSet中的任何DataTable对象中是否存在错误
	IsInitialized	Public	只读属性，表示是否初始化DataSet
	DefaultViewManager	Public	只读属性，获取DataSet所包含数据的自定义视图，以允许使用自定义的DataViewManager进行筛选、搜索和导航
	Relations	Public	只读属性，获取将表链接起来并允许从父表浏览到子表的关系的集合（数据库关系集合）
	EnforceConstraints	Public	读写属性，表示在尝试执行任何更新操作时是否遵循约束规则
	ExtendedProperties	Public	只读属性，获取与DataSet相关的自定义用户信息的集合
	Namespace	Public	读写属性，表示DataSet的命名空间

续表

分类	名　称	访问性	说　明
方法	DataSet	Public	构造函数，创建具有指定属性的DataSet对象
	HasChanges	Public	获取一个值，该值指示DataSet是否有更改，包括新增行、已删除的行或已修改的行
	GetChanges	Public	获取DataSet中被修改过的数据记录的集合副本
	AcceptChanges	Public	提交当前数据集合中所有未接受或拒绝的更改
	RejectChanges	Public	回滚当前数据集合中所有未接受或拒绝的更改
	BeginInit	Public	开始初始化在窗体上使用或由另一个组件使用的DataSet
	EndInit	Public	结束在窗体上使用或由另一个组件使用的DataSet的初始化
	Clear	Public	清除DataSet中的所有数据，通过清除所有表中的所有行实现
	Reset	Public	将DataSet重置为其初始状态
	Copy	Public	复制当前DataSet的结构和数据
	Merge	Public	将新的数据合并到当前DataSet或DataTable中，这些数据可以是一个DataSet、DataTable或DataRow的数组
	CreateDataReader	Public	为每个DataTable创建一个带有结果集的DataTableReader对象，可以只读向前访问数据表中的记录
	Load	Public	通过所提供的IDataReader，用某个数据源的值填充DataSet
	GetXml	Public	返回存储在DataSet中数据的XML表示形式
	GetXmlSchema	Public	返回存储在DataSet中数据的XML表示形式的XML架构
	InferXmlSchema	Public	将XML架构应用于DataSet
	ReadXml	Public	将XML架构和数据读入DataSet
	ReadXmlSchema	Public	将XML架构读入DataSet
	WriteXml	Public	往DataSet写XML数据，还可以选择写架构
	WriteXmlSchema	Public	写XML架构形式的DataSet结构

从表 11-4 中可以看出，DataSet 类的一些成员函数从名称到功能上都与 DataTable 相似，如 AcceptChanges()、RejectChanges()等，这些方法主要是通过 DataSet 中的各个 DataTable 执行对应函数完成，比如 DataSet 类的 AcceptChanges()方法，就是依次执行 DataSet 中所有 DataTable 的 AcceptChanges()方法实现。

和 DataTable 相比，DataSet 类的重点在于数据表（DataTable）的管理，比如 Copy()、Merge()等方法实现对数据表中数据记录的复制、合并等。所以 DataSet 中所有的数据表都保存在 Tables 属性中，通过该属性可以遍历 DataSet 中所有的 DataTable，而数据表之间的关系保存在 Relations 属性中。

11.3.2　管理 DataTable 集合

在使用 DataSet 之前，首先要创建 DataSet 对象，可以通过 DataSet 类的构造函数创建 DataSet 对象实例，DataSet 构造函数具有两个常用版本，定义如下。

- DataSet()：创建一个不带任何参数的 DataSet 对象，数据集的名称为默认值 NewDataSet。
- DataSet(string name)：创建一个具有指定名称的 DataSet 对象，参数 name 指定新数据集的名称。

DataSet 的一个主要目的就是在内存中模拟一个轻量级的关系型数据库，所以它包含并管理着一批数据表（DataTable），这些数据表都保存在它的 Tables 属性中，Tables 属性是一个 DataTableCollection 类型的只读属性，包含一个 DataTable 集合，可以通过它添加、删除、获取 DataTable 元素。

DataTableCollection 类型表示一个 DataTable 的集合，每个 DataTable 都有一个唯一的名称，同时还提供了多种管理 DataTable 的方法，包括添加、移除等。DataTableCollection 主要包括以下几个方法用来管理数据表。

- this 索引器：用来获取指定索引（从 0 开始）或指定名称的 DataTable。
- Add()：新建一个 DataTable 到该集合中，可以指定 DataTable 的名称。
- Remove()：从该集合中移除指定索引（从 0 开始）或指定名称的 DataTable。
- Clear()：从该集合中移除所有的 DataTable。
- Contains()：判断集合中是否包含指定名称的 DataTable。

示例代码 11-5 演示在 DataSet 中通过 Tables 属性管理 DataTable。CreateDataSet()方法首先创建一个名为 StudentScore 的 DataSet 对象，然后，依次创建两个名为 Students 和 Lessons 的 DataTable 对象，并将它们添加到 DataSet 中。最后，通过 RemoveAt()方法移除索引为 0（第 1 个）的 DataTable 对象。

示例代码 11-5

```
static void Main(string[] args)
{
    CreateDataSet();
}
static DataSet CreateDataSet()
{
    DataSet ds = new DataSet("StudentScore");     //创建名为 StudentScore
                                                     的数据集

    DataTable dt = ds.Tables.Add("Students");     //创建名为 Students 的数据表
    dt.Columns.AddRange(new DataColumn[] {         //创建 Students 数据表列信息
        new DataColumn("StudentID", typeof(int)),
        new DataColumn("StudentName", typeof(string)),
        new DataColumn("Password", typeof(string)),
        new DataColumn("Age", typeof(int)),
    });
    System.Console.WriteLine("1. DataSet 有{0}个 DataTable", ds. Tables.
    Count);
    dt = ds.Tables.Add("Lessons");                 //创建名为 Lessons 的数据表
    dt.Columns.AddRange(new DataColumn[] {         //创建 Lessons 数据表列信息
        new DataColumn("LessonID", typeof(int)),
        new DataColumn("LessonName", typeof(string)),
    });
    System.Console.WriteLine("2. DataSet 有{0}个 DataTable", ds.Tables.
    Count);

    System.Console.WriteLine("3. DataTable 的详细信息:");
    foreach (DataTable dtItem in ds.Tables)        //通过 Tables 遍历所有数据表
    {
```

```
        DisplayDataTable(dtItem);
    }

    ds.Tables.RemoveAt(0);                          //移除索引为 0 的 Data Table
    System.Console.WriteLine("4. DataSet 有{0}个 DataTable", ds.Tables.
    Count);
    return ds;
}
static void DisplayDataTable(DataTable dt)
{
    System.Console.WriteLine("数据表[{0}]有[{1}]个列", dt.TableName,
    dt.Columns.Count);
}
```

示例代码 11-5 的输出如下，从中可以看出，通过 DataSet 的 Tables 属性可以添加、删除 DataSet 中的 DataTable，同样也可以遍历 DataTable 信息。在获取到 DataTable 信息之后，可以根据业务逻辑需要，对它执行任何操作。

```
1. DataSet 有 1 个 DataTable
2. DataSet 有 2 个 DataTable
3. DataTable 的详细信息:
数据表[Students]有[4]个列
数据表[Lessons]有[2]个列
4. DataSet 有 1 个 DataTable
```

注意：在使用 DataTableCollection 的 Remove()和 RemoveAt()方法时，要先保证该集合中存在指定名称或索引的 DataTable，否则会产生异常。可以先通过 Contains()方法判断要删除的 DataTable 是否存在，如果存在则移除。

11.3.3　管理 DataTable 之间的关系

在关系数据库中，多个数据表之间往往存在各种关系，DataSet 作为内存中的关系数据库，同样可以维护并管理 DataTable 之间的关系。在 DataSet 中，通过 Relations 属性管理各 DataTable 之间的关系，Relations 属性为 DataRelationCollection 类型。

DataRelationCollection 和 DataTableCollection 类似，只是它用来管理一个或多个 DataRelation 元素，每个 DataRelation 对象表示一个数据表之间的关系，集合中每个关系具有一个唯一的名称。同时，DataRelationCollection 还提供了多种管理 DataRelation 的方法，包括添加、移除等。

- this 索引器：用来获取指定索引（从 0 开始）或指定名称的 DataRelation。
- Add()：新建一个 DataRelation 到该集合中，可以指定 DataRelation 的名称、外键、主键等信息。
- Remove()：从该集合中移除指定索引（从 0 开始）或指定名称的 DataRelation。
- Clear()：从该集合中移除所有的 DataRelation。
- Contains()：判断集合中是否包含指定名称的 DataRelation。

数据表之间的一个关系表示一个数据表对另外一个数据表之间的依赖关系，其中被依赖的数据表称为父表（Parent Table），另外一个数据表称为子表，父表和子表之间通过特

定的列关联起来。在创建一个 DataRelation 对象时，可以指定相互依赖的 DataColumn 对象，也可以指定相互依赖的 DataTable 的名称及对应的 DataColumn 的名称。

示例代码 11-6 演示如何创建 DataRelation。CreateRelationDataSet()方法首先创建一个名为 StudentScore 的 DataSet 对象，然后依次创建三个名为 Students、Lessons 和 Scores 的 DataTable 对象，并将它们添加到 DataSet 中。然后，添加两个关系到 Relations 属性中，最后打印所有关系的详细信息。

示例代码 11-6

```
static DataSet CreateRelationDataSet( )
{
    DataSet ds = new DataSet("StudentScore");    //创建名为 StudentScore 的数
                                                   据集
    DataTable dt = ds.Tables.Add("Students");   //创建名为 Students 的数据表
    dt.Columns.AddRange(new DataColumn[] {      //创建 Students 数据表的列信息
        new DataColumn("StudentID", typeof(int)),
        new DataColumn("StudentName", typeof(string)),
        new DataColumn("Password", typeof(string)),
        new DataColumn("Age", typeof(int)),
    });
    dt = ds.Tables.Add("Lessons");              //创建名为 Lessons 的数据表
    dt.Columns.AddRange(new DataColumn[] {      //创建 Lessons 数据表的列信息
        new DataColumn("LessonID", typeof(int)),
        new DataColumn("LessonName", typeof(string)),
    });
    dt = ds.Tables.Add("Scores");               //创建名为 Scores 的数据表
    dt.Columns.AddRange(new DataColumn[] {      //创建 Scores 数据表的列信息
        new DataColumn("StudentID", typeof(int)),
        new DataColumn("LessonID", typeof(int)),
        new DataColumn("Score", typeof(float)),
    });

    ds.Relations.Add("Students_Scores_Relation",    //创建Students 与Scores
                                                      之间的关系
                  ds.Tables["Students"].Columns["StudentID"],
                  ds.Tables["Scores"].Columns["StudentID"],
                  true);
    ds.Relations.Add("Lessons_Scores_Relation",     //创建 Lessons 与 Scores
                                                      之间的关系
                 ds.Tables["Lessons"].Columns["LessonID"],
                 ds.Tables["Scores"].Columns["LessonID"],
                 true);
    System.Console.WriteLine("数据集[{0}]共有{1}个关系",  //打印关系个数
                        ds.DataSetName,
                        ds.Relations.Count);
    foreach (DataRelation relation in ds.Relations)  //遍历并打印关系详细信息
    {
        System.Console.WriteLine("关系[{0}]详细信息:", relation.
```

```
            RelationName);
        System.Console.WriteLine("\t 数据集合: {0}", relation.
        DataSet.DataSetName);
        System.Console.WriteLine("\t 父表名称: {0}", relation.
        ParentTable.TableName);
        System.Console.WriteLine("\t 父表列名: {0}", relation.
        ParentColumns[0].ColumnName);
        System.Console.WriteLine("\t 子表名称: {0}", relation.
        ChildTable.TableName);
        System.Console.WriteLine("\t 子表列名: {0}", relation.
        ChildColumns[0].ColumnName);
    }
    return ds;
}
```

示例代码 11-6 的输出如下，从中可以看出，通过遍历 Relations 属性得到 DataSet 中所有的关系信息。在获得 DataRelation 对象之后，可以对它执行任何允许的操作。

```
数据集[StudentScore]共有 2 个关系
关系[Students_Scores_Relation]详细信息:
        数据集合: StudentScore
        父表名称: Students
        父表列名: StudentID
        子表名称: Scores
        子表列名: StudentID
关系[Lessons_Scores_Relation]详细信息:
        数据集合: StudentScore
        父表名称: Lessons
        父表列名: LessonID
        子表名称: Scores
        子表列名: LessonID
```

注意：通过 Relations 属性还可以删除数据表之间的关系，但是在删除之前要先通过 Contains()方法判断删除的关系是否存在，如果移除一个不存在的关系会产生异常。

11.3.4　提交和回滚 DataSet 的更改

DataSet 中数据的更改也只是在内存数据中的更改，和 DataTable 一样，可以通过 AccepteChanges()方法提交 DataSet 中所有数据表的所有更改，也可以通过 RejectChanges()方法回滚 DataSet 中所有数据表的所有更改。

在一些应用场景中，不需要提交或回滚 DataSet 中所有的数据记录更改，则可以通过 DataSet 的 Tables 属性获取要接受或回滚的 DataTable，通过 11.2.4 节介绍的方法提交或回滚某个数据表的更改。执行完成后 DataSet 类的 HasChanges()方法的返回值根据实际情况返回 true 或 false，因为其他表可能还包含更改记录。

在示例代码 11-7 中，AcceptAllChanges()方法通过 DataSet 类的 AcceptChanges()方法，提交 DataSet 中的所有更改记录。RejectAllChanges()方法通过 DataSet 类的 RejectChanges()方法，回滚 DataSet 中的所有更改记录。AcceptPartChanges()方法，首先通过 DataTable 类

的 AcceptChanges()方法接受 Students 表的更改记录，然后通过 DataTable 类的 RejectChanges()方法回滚 Lessons 表的更改记录。

示例代码 11-7

```
static void AcceptAllChanges(DataSet ds)
{
    //通过 DataSet 类的 AcceptChanges()方法接受所有更改
    ds.AcceptChanges( );
}
static void RejectAllChanges(DataSet ds)
{
    //通过 DataSet 类的 AcceptChanges()方法接受所有更改
    ds.RejectChanges( );
}
static void AcceptPartChanges(DataSet ds)
{
    DataTable dt;
    //通过 DataTable 类的 AcceptChanges()方法接受表 Students 的更改
    dt = ds.Tables["Students"];
    dt.AcceptChanges( );
    //通过 DataTable 类的 RejectChanges()方法回滚表 Lessons 的更改
    dt = ds.Tables["Lessons"];
    dt.RejectChanges( );
}
```

注意：DataSet 中任何一个数据表含有未提交的更改，它的 HasChanges()方法都会返回 true，所以在提交或回滚之前可以先通过 HasChanges()判断是否包含未提交的更改。

11.3.5　通过 DataSet 与 XML 交互

在 ADO.NET 中，DataSet 表示的内存关系数据库，可以通过 WriteXML()方法直接以 XML 格式写入到指定的文件中，也可以通过 ReadXML()方法从 XML 文件读取数据到数据集 DataSet 中。

WriteXML()方法包括多个重载版本，可以将 DataSet 中的数据保存到指定文件、指定的 TextWriter、指定的 XMLWriter 中，还可以指定写入时数据的映射模式。生成的 XML 数据会按照数据表依次列出，每个数据表具有一个独立的 XML 结点，该表的记录作为独立的子结点保存在表的结点下。

如示例代码 11-8 所示，DataSetToXML()方法首先创建一个名为 StudentScore 的 DataSet 对象，它包含两个数据表 Students 和 Lessons，这两个表各包含两条记录。最后，通过 WriteXML()方法将 DataSet 中的数据按照 XML 格式写入到文件 C:\StudentScore.XML 中。XMLToDataSet()方法正好相反，它通过 ReadXML()方法从文件 C:\StudentScore.XML 中读取数据到 DataSet 对象中，并打印出提示信息。

示例代码 11-8

```
static void Main(string[] args)
{
```

```
    DataSetToXML( );
    XMLToDataSet( );
}
static void DataSetToXML( )
{
    DataSet ds = new DataSet("StudentScore");  //创建名为 StudentScore 的数据集
    DataTable dt = ds.Tables.Add("Students");    //创建名为 Students 的数据表
    dt.Columns.AddRange(new DataColumn[] {        //创建 Students 数据表列信息
        new DataColumn("StudentID", typeof(int)),
        new DataColumn("StudentName", typeof(string)),
        new DataColumn("Password", typeof(string)),
        new DataColumn("Age", typeof(int)),
    });
    dt = ds.Tables.Add("Lessons");                //创建名为 Lessons 的数据表
    dt.Columns.AddRange(new DataColumn[] {        //创建 Lessons 数据表列信息
        new DataColumn("LessonID", typeof(int)),
        new DataColumn("LessonName", typeof(string)),
    });

    ds.Tables["Students"].Rows.Add(11, "李四", "lisi", 20);
                                                  //添加两个学生信息
    ds.Tables["Students"].Rows.Add(12, "王五", "wangwu", 22);

    ds.Tables["Lessons"].Rows.Add(21, "语文");    //添加两门功课信息
    ds.Tables["Lessons"].Rows.Add(22, "数学");
    ds.WriteXml(@"C:\StudentsScore.xml");         //保存为 XML 格式到文件
}

static void XMLToDataSet( )
{
    DataSet ds = new DataSet( );                  //创建 DataSet 对象
    ds.ReadXml(@"C:\StudentsScore.xml");          //从 XML 文件读取数据
    System.Console.WriteLine("数据集[{0}]具有{1}个表.", ds.DataSetName,
ds.Tables.Count);
    foreach (DataTable dt in ds.Tables)           //打印所有数据表的信息
    {
        System.Console.WriteLine("数据表[{0}]有[{1}]行数据.", dt.TableName,
        dt.Rows.Count);
    }
}
```

执行示例代码 11-8，可以从 C 盘找到新生成的 XML 文件 StudentScore.XML，打开可以看到它的内容如下所示。从中可以看出，DataSet 名称作为 XML 文件的根（StudentScore），DataSet 中各个 DataTable 作为根元素的子元素。值得注意的是，DataTable 的每条记录都会出现一个对应的结点，例如，这里出现了两个 Students 和 Lessons 结点，因为它们各有两条记录。

```
<?xml version="1.0" standalone="yes"?>
<StudentScore>
  <Students>
    <StudentID>11</StudentID>
    <StudentName>李四</StudentName>
```

```
    <Password>lisi</Password>
    <Age>20</Age>
  </Students>
  <Students>
    <StudentID>12</StudentID>
    <StudentName>王五</StudentName>
    <Password>wangwu</Password>
    <Age>22</Age>
  </Students>
  <Lessons>
    <LessonID>21</LessonID>
    <LessonName>语文</LessonName>
  </Lessons>
  <Lessons>
    <LessonID>22</LessonID>
    <LessonName>数学</LessonName>
  </Lessons>
</StudentScore>
```

11.4　小　　结

ADO.NET 是.NET 4.0 中进行数据库处理的一个重要技术，它通过不同的数据提供程序从不同的数据库获取数据，并将数据保存在内存中，所以说数据提供程序是 ADO.NET 实现多数据库支持的核心部分之一。

ADO.NET 通过在内存中模拟简单的关系数据库，实现在内存中对数据库中的数据进行修改，主要包括两个核心类 DataSet 和 DataTable。DataTable 类表示一个数据表，用来表示数据表的列定义信息、关系等，最主要是保存数据记录。DataSet 类表示一个数据集，类似一个简单的内存数据库，它主要负责多个数据表的管理。

通过本章的学习，读者应该掌握以下知识点：

- 什么是 ADO.NET？
- 数据提供程序在 ADO.NET 中的作用和地位如何？
- DataTable 类具有哪些成员，各有什么功能？
- 如何管理 DataTable 的列定义？
- 如何管理 DataTable 的数据记录？
- 如何提交和回滚 DataTable 中数据的更改？
- 如何监视 DataTable 的更改？
- DataSet 类具有哪些成员，各有什么功能？
- 如何管理 DataSet 中的 DataTable 集合？
- 如何管理 DataSet 中多个 DataTable 之间的关系？
- 如何提交和回滚 DataSet 中数据的更改？
- 如何将 DataSet 中的数据保存为 XML 格式？
- 如何从 XML 文件中读取数据到 DataSet 中？

第 12 章　使用 ADO.NET 访问数据库

访问数据库是 ADO.NET 存在的最终目的，ADO.NET 首先从数据库中获取数据到内存，然后再对内存中的数据进行处理，最后更新到数据库中。ADO.NET 除了直接处理内存数据，还可以直接执行 SQL 命令，对数据库进行更新。第 11 章介绍了如何在内存中对数据进行处理，本章将介绍如何通过 ADO.NET 与数据库进行交互。

12.1　了解 ADO.NET 访问数据库原理

ADO.NET 将访问数据库的操作分成多个可以分解的独立动作，而且支持有连接和无连接两种访问模式，本节将简单介绍 ADO.NET 进行数据库交互所需要的主要类和概念。

12.1.1　了解 ADO.NET 数据库操作类

在 ADO.NET 中，数据库访问被分解成多个独立的部分，每个部分都用一个独立的类封装起来，各个部分完成各自的功能，比如数据库连接、数据查询命令、数据读取器等。在 ADO.NET 中，所有的核心类都包含在命名空间 System.Data.Common 中，包括以下几个主要的类。

- DbConnection 类：表示一个与数据库服务器之间的连接，它是所有数据库连接类的基类，它提供了打开和关闭数据库连接、执行事务、创建命令等方法。DbConnection 类包括一个连接字符串（ConnectionString）属性，该属性描述了数据库服务器的连接信息，包括服务器地址、登录用户名和密码、目标数据库等。
- DbCommand 类：表示一个可以执行的 SQL 命令，可以是 SELECT、DELETE、UPDATE 等通用的 SQL 命令，也可以是执行存储过程命令等高级 SQL 命令。DbCommand 类接受一个表示 SQL 命令的字符串，并可以在它连接的 DbConnection 对象上执行该命令。
- DbParameter 类：表示 SQL 命令中的一个参数，包括参数名、参数类型等信息。DbCommand 类通过 DbParameter 类来表示 SQL 命令中的参数，并且将参数的值组合到 SQL 命令，从而正确执行。
- DbDataReader 类：表示一个只读的向前的数据读取器，通过它可以从第一条记录开始，读取 SQL 命令产生的所有记录。DbDataReader 对象通常由 DbCommand.ExecuteReader()方法产生，而且 DbCommand 中通常是一个 SELECT 查询命令，也可以是一个返回数据集的存储过程。
- DbDataAdapter 类：表示一个数据库适配器，它通过 DbCommand 类执行 SELECT

命令，从数据库服务器获取查询结果，并填充到内存数据集 DataSet 中。当内存 DataSet 中的数据更改后，DbDataAdapter 提交这些更新到数据库。

ADO.NET 通过上面这几个核心类来完成最基本的数据库操作，并辅助 DbTransaction 来完成事务的执行。不同的数据提供程序通常需要继承并实现这些类，完成与目标数据库进行交互的功能。ADO.NET 内置了 4 种数据提供程序，分别用于访问 Microsoft SQL Server、Access、ODBC、Oracle 4 种数据库，这些数据提供程序继承并实现了前面提到的基类，如表 12-1 所示。

表 12-1　ADO.NET数据提供程序和对应的类

	基类	SQL Server	OleDb	ODBC	Orcale
命名空间	System.Data.Common	System.Data.SqlClient	System.Data.OleDb	System.Data.Odbc	System.Data.OracleClient
连接类	DbConnection	SqlConnection	OleDbConnection	OdbcConnection	OracleConnection
命令类	DbCommand	SqlCommand	OleDbCommand	OdbcCommand	OracleCommand
阅读器类	DbDataReader	SqlDataReader	OleDbDataReader	OdbcDataReader	OracleDataReader
适配器类	DbDataAdapter	SqlDataAdapter	OleDbDataAdapter	OdbcDataAdapter	OracleDataAdapter
参数类	DbParameter	SqlParameter	OleDbParameter	OdbcParameter	OracleParameter

从表 12-1 中可以看出，ADO.NET 各数据提供程序实现的类在命名规则上都非常有规律，所以使用起来非常方便。开发人员在实现特定的数据提供程序时也应该尽可能遵循这个命名规则。

技巧：OleDb 数据提供程序可以用来访问 ACCESS 数据库，也可以访问 SQL Server 数据库。但是 SQL Server 数据提供程序对 SQL Server 数据库提供了更加全面和完整的支持，应该尽可能用它来访问 SQL Server 数据库。

12.1.2　两种 ADO.NET 访问数据库的模式

ADO.NET 提供两种模式来访问数据库：连接模式和无连接模式。ADO.NET 在连接模式下访问数据库时，在取得数据库连接之后，保持数据库连接，通过向数据库服务器发送 SQL 命令等方式实时更新到数据库。在连接模式下的数据库访问通常包括以下几个步骤：

（1）通过数据库连接类（DbConnection）连接到指定数据库服务器的数据库。

（2）通过数据库命令类（DbCommand）在数据库上执行 SQL 命令，可以是任何 SQL 命令，包括更新（UPDATE）、插入（INSERT）、删除（DELETE）、查询（SELECT）等。

（3）如果是查询（SELECT）语句，可以通过数据读取器（DbDataReader）类只读只向前读取数据记录，并对数据库记录进行处理。

（4）数据库操作完成后通过数据库连接类（DbConnection）关闭数据库连接，释放占用的资源。

在连接模式下访问数据库时，客户端和数据库服务器之间一直保持连接，所以尽量不要长时间维持连接，因为这样会导致数据库服务器被长期占用，影响其他客户端连接到数据库服务器。笔者建议，在使用之前打开数据库连接，使用完成后马上关闭数据库连接。

当需要对数据进行长时间处理时，通常采用无连接模式进行数据访问。在 ADO.NET

无连接模式下，需要处理的数据库服务器中的数据在本地有一个副本，通常保存在 DataSet 或 DataTable 中，ADO.NET 通过数据适配器（DbDataAdapter）将内存数据集（DataSet）和数据库服务器中的数据关联起来。从数据库服务器取得数据之后，数据适配器断开与服务器的连接，对数据的修改都通过修改内存 DataSet 完成，然后再通过数据适配器更新到服务器。在 ADO.NET 中，无连接模式的数据库访问通常需要以下步骤：

（1）通过数据库连接类（DbConnection）连接到指定数据库服务器的数据库。

（2）创建基于该数据库连接的数据适配器（DbDataAdapter），并指定访问数据库的 SQL 命令，包括插入（INSERT）、更新（UPDATE）、查询（SELECT）和删除（DELETE）4 个命令。数据适配器（DbDataAdapter）通过这几个命令从数据库获取数据，也将本地的数据更改更新到数据库服务器。

（3）通过数据适配器（DbDataAdapter）从数据库服务器获取数据到 DataSet 或 DataTable 中。

（4）使用或更改 DataSet 或 DataTable 中的数据。

（5）通过数据适配器（DbDataAdapter）将 DataSet 数据的更改提交到数据库服务器，并关闭数据库连接。

基于无连接的数据库访问，具有执行效率高、数据库连接占用时间短、修改记录容易提交和回滚等优点。但是在一定程度上会导致数据更新不及时，所以需要更多考虑数据同步问题。

本章后面几节将介绍如何通过 ADO.NET 提供的 SQL Server 数据提供程序访问 SQL Server 数据库，其他类型数据库的访问完全类似，不再赘述。

12.2　ADO.NET 连接模式访问数据库

ADO.NET 连接模式访问数据库的关键是连接到数据库、创建 SQL 命令和执行 SQL 命令，本节将介绍如何通过 ADO.NET 连接模式访问数据库。

12.2.1　了解 SqlConnection 连接类

在 ADO.NET 中，SqlConnection 类表示与 SQL Server 数据库的一个唯一会话，它通过指定的数据库连接字符串，连接到数据库，并打开数据库。表 12-2 给出了 SqlConnection 类的常用成员。

表 12-2　SqlConnection类常用成员

分类	名　称	访问性	说　明
属性	ConnectionString	Public	读写属性，表示用于打开SQL Server数据库的连接字符串，包括数据库服务器的IP地址、端口、目标数据库、安全性等信息
	ConnectionTimeout	Public	只读属性，表示尝试连接到数据库服务器判断为连接失败的等待时间，单位秒，由连接字符串指定
	Database	Public	只读属性，表示当前数据库或连接打开后要使用的数据库的名称，由连接字符串指定

续表

分类	名　称	访问性	说　明
属性	DataSource	Public	只读属性，表示要连接的SQL Server实例的名称
	PacketSize	Public	只读属性，表示用来与SQL Server实例通信的网络数据包的大小，单位字节
	ServerVersion	Public	只读属性，获取包含客户端连接的SQL Server实例的版本
	State	Public	只读属性，表示当前数据库连接的状态
	WorkstationId	Public	只读属性，获取标识数据库客户端的一个字符串
	StatisticsEnable	Public	读写属性，表示是否进行统计信息收集
方法	Open	Public	使用ConnectionString所指定的属性打开数据库连接
	Close	Public	关闭与数据库的连接
	BeginTransaction	Public	开始在SQL Server数据库上执行一个事务
	ChangeDatabase	Public	为打开的数据库连接更改当前数据库
	ChangePassword	Public	将连接字符串中指定用户的SQL Server密码更改为提供的新密码
	CreateCommand	Public	创建并返回一个与SqlConnection关联的SqlCommand对象
	ClearAllPools	Public	清空连接池
	ClearPool	Public	清空与指定连接关联的连接池

在表 12-2 列出的众多属性和方法中，在数据库连接和断开操作中最常用的有 3 个。

- ConnectionString：连接字符串，它包含数据库服务器的地址、端口、目标数据库、连接超时时间、安全性、登录用户名和密码等信息。在进行数据连接之前，必须指定正确的连接字符串。
- Open()：用于打开由 ConnectionString 属性指定的数据库连接，如果连接字符串不正确，或目标服务器不可用（比如没有打开，不存在等）都会抛出异常。
- Close()：关闭一个已经打开的数据库连接，如果当前并没有连接，则不做任何操作。

连接字符串（ConnectionString）是连接到数据库的一个核心元素，它定义了数据库服务器的地址、数据库名称、登录名和密码等连接信息。只有在数据库连接关闭时才能设置连接字符串的值。

连接字符串的基本格式包括一系列由分号分隔的键/值对，等号（=）用来连接各个关键字及其值，键或值里面如果有等号，则需要用两个等号表示这个等号。若要在字符串值中包括前导或尾随空格，则该值必须用单引号或双引号括起来。即使将整数、布尔值或枚举值用引号括起来，其周围的任何前导或尾随空格也将被忽略，但保留字符串关键字或值内的空格。表 12-3 列出了 SQL Server 连接字符串常用的键及其默认值。

表 12-3　SQL Server连接字符串常用的关键字

键	默　认　值	说　明
AttachDBFilename 或extended properties 或Initial File Name	N/A	主数据库文件的名称，包括可连接数据库的完整路径名。只有具有“.mdf”扩展名的主数据文件才支持AttachDBFilename
Connect Timeout 或Connection Timeout	15	在终止尝试并产生错误之前，等待与服务器的连接的时间长度（以秒为单位）

续表

键	默　认　值	说　　明
Current Language	N/A	SQL Server 语言记录名称
Data Source 或Server 或Address 或Addr 或Network Address	N/A	要连接的 SQL Server 实例的名称或网络地址。可以在服务器名称之后指定端口号，指定本地实例时，始终使用（local）若要强制使用某个协议，请添加下列前缀之一：np:(local), tcp:(local), lpc:(local)
Encrypt	'false'	当该值为true时，如果服务器端安装了证书，则SQL Server将对所有在客户端和服务器之间传送的数据使用SSL加密。可识别的值为true、false、yes和no
Initial Catalog 或Database	N/A	数据库的名称
Integrated Security 或Trusted_Connection	'false'	当为false时，将在连接中指定用户ID和密码。当为true时，将使用当前的 Windows账户凭据进行身份验证
Packet Size	8192	用来与 SQL Server 的实例进行通信的网络数据包的大小，以字节为单位
Password 或Pwd	N/A	SQL Server 账户登录的密码
Type System Version	N/A	指示应用程序期望的类型系统的字符串值。可能的值包括： Type System Version=SQL Server 2000; Type System Version=SQL Server 2005; Type System Version=SQL Server 2008; Type System Version=Latest;（表示最新版本）
User ID	N/A	SQL Server登录账户
Workstation ID	本地计算机名称	连接到 SQL Server 的工作站的名称

注意：ADO.NET 相关的类库保存在 System.Data 命名空间下，而 SQL Server 数据库访问的类库放在 System.Data.SqlClient 命名空间下。所以，代码中必须引用该命名空间，如下所示。

```
using System.Data;
using System.Data.SqlClient;
```

12.2.2　用 SqlConnection 创建数据库连接

在 ADO.NET 中，通过 SqlConnection 类进行数据库连接管理，通常包括设置连接字符串、打开链接和关闭连接 3 个主要操作。具体需要执行以下几个步骤：

（1）通过构造函数创建一个 SqlConnection 对象，可以同时指定连接字符串。

（2）如果在第 1 步中没有设置连接字符串，则要设置正确的连接字符串。

（3）通过 SqlConnection.Open()方法打开数据库连接。

（4）操作完成后，通过 SqlConnection.Close()方法关闭数据库连接。

在示例代码 12-1 中，CreateConnection()方法演示 SqlConnection 类的使用，按照上面

介绍的 4 个步骤打开数据库连接，最后关闭连接。通过表 12-2 中介绍的 SqlConnection 类的属性来获取数据库连接的详细信息，并打印出这些信息。其中，连接字符串 conStr 指定了数据库服务器（Data Source）是 WWW-818324B7DD9\YMYSQLSERVER，要访问的数据库(Initial Catalog)是 UserLog，Integrated Security 关键字为 true 表示使用内置的 Windows 认证。

示例代码 12-1

```
static void Main(string[] args)
{
    CreateConnection( );
}

static void CreateConnection( )
{
    //连接字符串，使用数据库 UserLog
    string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
    Catalog=UserLog;Integrated Security= true ";
    //用连接字符串 conStr 创建 SqlConnection 对象 connection
    SqlConnection connection = new SqlConnection(conStr);
    //打印打开连接前 SqlConnection 的默认信息
    System.Console.WriteLine("数据库连接--Open()前信息:");
    System.Console.WriteLine("  ConnectionTimeout=[{0}]", connection.
    ConnectionTimeout);
    System.Console.WriteLine("  Database=[{0}]", connection.Database);
    System.Console.WriteLine("  DataSource=[{0}]", connection.
    DataSource);
    System.Console.WriteLine("  PacketSize=[{0}]", connection.
    PacketSize);
    System.Console.WriteLine("  StatisticsEnabled=[{0}]", connection.
    StatisticsEnabled);
    System.Console.WriteLine("  WorkstationId=[{0}]", connection.
    WorkstationId);
    System.Console.WriteLine("  State=[{0}]", connection.State);
    //打开数据库连接
    connection.Open( );
    //打印打开连接后 SqlConnection 的信息
    System.Console.WriteLine("数据库连接--Open()后信息:");
    System.Console.WriteLine("  ServerVersion=[{0}]", connection.
    ServerVersion);
    System.Console.WriteLine("  State=[{0}]", connection.State);
    //关闭数据库连接
    connection.Close( );
    //打印关闭连接后 SqlConnection 的信息
    System.Console.WriteLine("数据库连接--Close()后信息:");
    System.Console.WriteLine("  State=[{0}]", connection.State);
}
```

示例代码 12-1 的输出如下，值得注意的是 SqlConnection.State（状态）的变化，默认为关闭（Closed）状态，打开连接之后变为打开（Open）状态，关闭连接之后又变为关闭（Closed）状态。另外，SqlConnection 类的 ServerVersion 属性只能在数据库连接打开时才能正确执行，如果数据库连接没有打开，则会抛出异常。

```
数据库连接--Open()前信息:
 ConnectionTimeout=[15]
```

```
  Database=[UserLog]
  DataSource=[WWW-818324B7DD9\YMYSQLSERVER]
  PacketSize=[8000]
  StatisticsEnabled=[ false]
  WorkstationId=[WWW-818324B7DD9]
  State=[Closed]
数据库连接--Open()后信息:
  ServerVersion=[09.00.1399]
  State=[Open]
数据库连接--Close()后信息:
  State=[Closed]
```

技巧：为了保证数据库连接在任何情况下都被关闭，通常使用 try…finally 语句，在 try 语句块中打开并使用 SqlConnection，在 finally 语句块中关闭 SqlConnection。

12.2.3　了解 SqlCommand 命令类

ADO.NET 在连接模式下进行数据库访问，可以通过 SqlCommand 类在 SQL Server 数据库上执行 SQL 命令。SqlCommand 类封装了要对 SQL Server 数据库执行的 Transact-SQL 语句或存储过程。最常用也是最简单的使用方式是，通过 SqlCommand 类直接执行表示 SQL 命令字符串。SqlCommand 类支持任何 SQL 命令语句的执行，表 12-4 列出了它的常用成员。

表 12-4　SqlCommand类常用成员

分类	名　　称	访问性	说　　明
属性	CommandText	Public	读写属性，表示要对数据源执行的Transact-SQL 语句或存储过程
	CommandTimeout	Public	读写属性，表示判断执行失败所需要的超时时间，单位秒
	CommandType	Public	读写属性，表示如何解释CommandText属性，通常为文本格式
	Connection	Public	读写属性，表示SqlCommand中要使用的数据库连接
	Parameters	Public	只读属性，表示当前SQL命令所需要的参数列表
	Transaction	Public	读写属性，将在其中执行SqlCommand的SQL Server过程
方法	BeginExecuteNonQuery	Public	启动一个异步操作，执行Transact-SQL语句或存储过程指定操作，通常为不返回数据集的操作，比如Update、Insert、Delete，返回被影响的数据记录的条数
	ExecuteNonQuery	Public	同步执行Transact-SQL语句或存储过程指定操作，通常为不返回数据集的操作，比如Update、Insert、Delete，返回被影响的数据记录的条数
	EndExecuteNonQuery	Public	停止BeginExecuteNonQuery()启动的异步操作
	BeginExecuteReader	Public	启动一个异步操作，执行Transact-SQL语句或存储过程指定的查询操作，比如Selecte语句，返回只读只向前的数据读取器
	ExecuteReader	Public	同步执行Transact-SQL语句或存储过程指定的查询操作，比如Selecte语句，返回只读只向前的数据读取器
	EndExecuteReader	Public	停止BeginExecuteReader()启动的异步操作
	ExecuteScalar	Public	同步执行一个查询操作，并返回查询所返回的结果集中第一行的第一列的数据，忽略其他列或行
	Cancel	Public	尝试取消SqlCommand的执行操作
	ResetCommandTimeout	Public	将CommandTimeout属性重置为其默认值

通过 SqlCommand 类更改数据库记录，通常分为两类操作：更新类和查询类。更新类操作不需要返回数据集，如 UPDATE、DELETE、INSERT 命令，通常通过 SqlCommand 类的 ExecuteNonQuery()方法执行这类命令，返回实际影响的数据记录的数量。查询类操作则需要返回数据，如 SELECT 命令，通常通过 SqlCommand 类的 ExecuteReadeer()方法执行命令，并返回一个只读只向前的 SqlDataReader 对象。

12.2.4　用 SqlCommand 类执行更新操作

在 ADO.NET 有连接模式下访问数据库，通常通过 SqlCommand 类执行指定的 SQL 命令，包括更新类操作和查询类操作两种。更新类操作主要用于更新数据库数据，通常只返回受到影响的记录数。通过 SqlCommand 类执行更新操作通常包括以下几个步骤。

（1）通过 SqlConnection 类建立可用的数据库连接。

（2）在可用数据库连接基础上创建一个 SqlCommand 对象。

（3）通过 SqlConnection.Open()方法打开数据库连接。

（4）通过 SqlCommand.CommandText 属性设置它要执行的 SQL 命令。

（5）使用 SqlCommand.ExecuteNoQuery()方法执行 SQL 命令，返回受到影响的记录数。

（6）如果需要，则重复第 4 步和第 5 步，执行更多 SQL 命令。

（7）通过 SqlConnection.Close()方法关闭数据库连接。

注意： 上面步骤中，第 4 步和第 5 步持续的时间不能太长，这样会长期占用数据库连接，容易造成服务器阻塞，如果需要长时间的数据库操作，可以考虑分成多个独立的小步来完成。

示例代码 12-2 演示上面 7 个步骤的使用实例，在 UpdateDataFunc()方法中，首先，创建一个数据库连接 connection，并且从 connection 创建一个 SqlCommand 对象 cmd，并打开数据库连接；然后，CommandText 属性设置要执行的 SQL 命令，并通过 ExecuteNonQuery()方法执行命令；最后，关闭数据库连接。

示例代码 12-2

```
static void Main(string[] args)
{
    UpdateDataFunc( );
}

static void UpdateDataFunc( )
{
    //连接字符串，使用数据库 UserLog
    string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
    Catalog=UserLog;Integrated Security=true";
    //用连接字符串 conStr 创建 SqlConnection 对象 connection
    SqlConnection connection = new SqlConnection(conStr);
    //通过 SqlConnection 创建 SqlCommand 对象
    SqlCommand cmd = connection.CreateCommand( );
    connection.Open( );                                          //打开数据库连接
    cmd.CommandText = "UPDATE Users SET Age=Age+1";              //设置 SQL 命令
    int rowCount = cmd.ExecuteNonQuery( );                       //执行 SQL 命令
```

```
    System.Console.WriteLine("1. {0}行记录被影响.", rowCount);
    //设置 SQL 命令
    cmd.CommandText = "UPDATE Logs SET LogTime='"+ DateTime.Now.ToString()
    +"' WHERE ID=1";
    rowCount = cmd.ExecuteNonQuery( );                          //执行 SQL 命令
    System.Console.WriteLine("2. {0}行记录被影响.", rowCount);
    connection.Close( );                                        //关闭数据库连接
}
```

示例代码 12-2 的输出如下，第 1 次 UPDATE 命令影响了 2 条记录，第 2 次 UPDATE 命令影响了 1 条记录。这里的输出可能会根据数据库的数据记录数不同而有所不同。

```
1. 2 行记录被影响.
2. 1 行记录被影响.
```

技巧： 通常有两种方法可以创建 SqlCommand 对象，一种是直接通过构造函数创建，另一种是通过 SqlConnection.CreateCommand() 方法创建，后者自动指定 SqlCommand 的 Connection 属性，不需要专门设置。

12.2.5　用 SqlDataReader 读取记录

在基于连接的数据库访问模式下，查询类操作通常是执行 SELECT 命令，产生的查询结果可以通过 SqlDataReader 类依次读取。SqlDataReader 类是 ADO.NET 提供的用于读取 SQL Server 数据库记录的只读只向前数据记录读取器。

通过 SqlCommand.ExecuteReader()方法执行 SQL 命令，执行完成后返回一个可以获取查询结果的 SqlDataReader 对象。开始时 SqlDataReader 指向第一条记录之前，不能直接，通过 SqlDataReader.Read()方法可以读取下一条记录，重复执行，直到全部记录读取完成。

为了方便获取数据记录中各字段的值，SqlDataReader 类还提供 GetXXXX()系列方法，将指定字段的数据按照特定数据类型读取，比如 int、string、DateTime 等。表 12-5 给出了 SqlDataReader 类的常用成员。

表 12-5　SqlDataReader类常用成员

分类	名　　称	访问性	说　　明
属性	Depth	Public	只读属性，表示当前行的嵌套深度
	FieldCount	Public	只读属性，表示当前行中的列数
	HasRows	Public	只读属性，表示当前SqlDataReader是否包含一行或多行
	IsClosed	Public	只读属性，表示当前SqlDataReader实例是否已经关闭
	VisibleFieldCount	Public	只读属性，表示当前SqlDataReader中未隐藏的字段的数目
方法	Read	Public	使当前SqlDataReader前进到下一条记录，即向前读取
	NextResult	Public	当读取批处理Transact-SQL语句的结果时，使数据读取器前进到下一个结果，注意并非下一行记录
	IsDBNull	Public	确定指定列中是否包含不存在或缺少的值，这只有在列数据允许为空时才可能为true
	Close	Public	关闭当前SqlDataReader实例，关闭后，IsClosed属性为true
	GetFieldType	Public	获取指定列的数据类型的Type
	GetDataTypeName	Public	获取源数据类型的名称

续表

分类	名　称	访问性	说　明
方法	GetName	Public	获取指定列的名称
	GetOrdinal	Public	在给定列名称的情况下获取列序号
	GetBoolean	Public	按照布尔（bool）类型读取指定列的数据
	GetByte	Public	按照字节（byte）类型读取指定列的数据
	GetBytes	Public	从指定的列偏移量将字节流读入缓冲区，并将其作为从给定的缓冲区偏移量开始的数组
	GetChar	Public	按照字符（char）类型读取指定列的数据
	GetChars	Public	从指定的列偏移量将字符流作为数组从给定的缓冲区偏移量开始读入缓冲区
	GetDateTime	Public	按照日期时间（DateTime）类型读取指定列的数据
	GetDecimal	Public	按照Decimal类型读取指定列的数据
	GetDouble	Public	按照双精度浮点数（double）类型读取指定列的数据
	GetFloat	Public	按照单精度浮点数（float）类型读取指定列的数据
	GetInt16	Public	按照16位整数（short）类型读取指定列的数据
	GetInt32	Public	按照32位整数（int）类型读取指定列的数据
	GetInt64	Public	按照64位整数（long）类型读取指定列的数据
	GetString	Public	按照字符串（string）类型读取指定列的数据

通过 SqlCommand 类和 SqlDataReader 类执行查询操作，通常需要以下几个步骤。

（1）通过 SqlConnection 类建立可用的数据库连接。

（2）在可用数据库连接基础上创建一个 SqlCommand 对象。

（3）通过 SqlConnection.Open()方法打开数据库连接。

（4）通过 SqlCommand.CommandText 属性设置它要执行的 SQL 命令。

（5）使用 SqlCommand.ExecuteReader()方法执行 SQL 命令，并返回 SqlDataReader 对象。

（6）通过 SqlDataReader.GetXXXX()方法读取某个字段的值。

（7）通过 SqlDataReader.Read()方法读取下一条记录，重复第 6 步直到记录全部读完。

（8）如果需要，则重复第 4 步到第 7 步，执行更多 SQL 命令。

（9）通过 SqlConnection.Close()方法关闭数据库连接。

注意：上面步骤中，第 4 步到第 7 步持续的时间不能太长，这样会长期占用数据库连接，容易造成服务器阻塞。通常在第 6 步只对数据进行简单的筛选和过滤，并将需要处理的数据读取并保存到内存中，然后再在内存中对数据进行复杂的处理。

示例代码 12-3 演示如何通过 SqlConnection 和 SqlDataReader 查询数据记录。首先，创建一个数据库连接 connection，并且从 connection 创建一个 SqlCommand 对象 cmd，并打开数据库连接；然后，CommandText 属性设置要执行的 SQL 命令，并通过 ExecuteReader()方法执行命令并获取 SqlDataReader 对象的 reader；之后，通过 reader 对象的 GetString()、GetInt32()等方法获取并打印记录的值；最后，关闭数据库连接。

示例代码 12-3

```
static void Main(string[] args)
{
    SelectDataFunc( );
}

static void SelectDataFunc( )
{
    //连接字符串，使用数据库 UserLog
    string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
    Catalog=UserLog;Integrated Security=true";
    //用连接字符串 conStr 创建 SqlConnection 对象 connection
    SqlConnection connection = new SqlConnection(conStr);
    //通过 SqlConnection 创建 SqlCommand 对象
    SqlCommand cmd = connection.CreateCommand( );
    connection.Open( );                                //打开数据库连接
    //设置 SQL 命令
    cmd.CommandText = "SELECT Name, Age, XingBie, Mobile FROM Users";
    SqlDataReader reader = cmd.ExecuteReader( );       //执行 SQL 命令
    System.Console.WriteLine("查询结果集如下：");
    while (reader.Read())                              //重复读取全部读完
    {
        //读取并打印 Name，索引为 0，string 类型
        System.Console.Write("{0}\t", reader.GetString(0));
        //读取并打印 XingBie，索引为 2，string 类型
        System.Console.Write("{0}\t", reader.GetString(2));
        //读取并打印 Age，索引为 1，int 类型
        System.Console.Write("{0}\t", reader.GetInt32(1));
        //读取并打印 Mobile，索引为 3，string 类型
        System.Console.Write("{0}\t", reader.GetString(3));
        System.Console.WriteLine( );
    }
    connection.Close( );                               //关闭数据库连接
}
```

示例代码 12-3 的输出如下，从中可以看出通过 SqlDataReader 可以轻松地读取全部数据查询结果，但是要注意的是 SqlDataReader 是只读且只向前的，也就是说不能返回到上一条记录。

```
查询结果集如下：
张三    男    23    131123456789
李花    女    24    133331234577
```

技巧： 通常通过“while(reader.Read()){ }”遍历所有的记录，另外 SqlDataReader. GetXXXX()方法中需要的索引是从 0 开始计数，而且根据 SELECT 命令判断各字段的索引。

12.3　执行带参数的 SQL 命令

在 12.2 节介绍的所有更新操作和查询操作中的 SQL 命令都不带参数，但是 SqlCommand 同样可以支持带参数的 SQL 命令，本节将介绍如何通过 SqlCommand 执行查

询带参数的 SQL 命令。

12.3.1　了解 SqlParameter 参数类

实际开发中，很多 SQL 命令都不只是简单的静态文本，往往包含一个或多个参数，在 SQL 命令中，参数是一个用@标记的变量，它可以是任何 SQL 数据类型。所以要进行数据库操作，带参数的 SQL 命令必不可少，这也是任何数据库访问组件必须支持的功能。

在 ADO.NET 中，SqlCommand 表示一个 SQL 命令，并封装了执行 SQL 命令的功能，它同时也支持带参数的 SQL 命令。SqlParameter 类表示一个 SQL 命令的参数，包括参数名、数据类型等信息。SqlCommand 类的 Parameters 属性（见表 12-4）是一个 SqlParameter 列表，通过它可以管理这个命令所需要的参数，包括添加和删除参数、设置参数的值等。

表 12-6 列出了 SqlParameter 类常用的成员，从中可以看出 SqlParameter 类的属性使用比较广泛，通常也是为它的属性指定特定的值即可。其中比较重要的属性包括如下几个，这也是在使用一个 SqlParameter 时必须指定的属性。

- ParameterName：表示该参数的名称，以“@参数名”的格式来表示。
- DbType 和 SqlDbType：表示该参数的数据类型，它是 SqlDbType 枚举类型，该枚举的各个值将 SQL Server 数据类型与.NET 数据类型相关联，它们之间的对应关系如表 12-7 所示。
- Value 和 SqlValue：表示该参数的值，该值的具体类型与 DbType 和 SqlDbType 相对应。

表 12-6　SqlParameter类常用成员

分类	名　　称	访　问　性	说　　明
属性	ParameterName	Public	读写属性，表示该参数的名称
	DbType	Public	读写属性，表示该参数在SQL Server数据库中类型
	SqlDbType	Public	读写属性，表示该参数在SQL Server数据库中类型
	Value	Public	读写属性，表示该参数的值
	SqlValue	Public	读写属性，表示该参数作为SQL Server类型的参数的值
	Offset	Public	读写属性，表示该参数对Value属性的偏移量
	IsNullable	Public	读写属性，表示该参数是否接受空值
	Direction	Public	读写属性，表示该参数是只可输入、只可输出、双向还是存储过程返回值参数
	Precision	Public	读写属性，表示该参数的值的最大位数
	Scale	Public	读写属性，表示该参数的值解析为的小数位数
	Size	Public	读写属性，表示该参数的值的最大大小（以字节为单位）
	CompareInfo	Public	读写属性，定义应该如何为此参数执行字符串比较
	TypeName	Public	读写属性，表示该参数的类型名称
方法	SqlParameter	Public	构造函数，创建一个指定的SqlParameter对象
	ResetDbType	Public	重置与该参数关联的类型
	ResetSqlDbType	Public	重置与该参数关联的类型

表 12-7　SqDbType枚举与C#数据类型

SqlDbType	.NET数据类型	说　明
BigInt	Int64	64位的有符号整数
Binary	Byte[]	二进制数据的固定长度流，范围在1～8000个字节之间
Bit	bool?	布尔值，可以是true、false或null
Char	String	非Unicode字符的固定长度流，范围在1～8000个字符之间
DateTime	DateTime	日期和时间数据，值范围从1753年1月1日到9999年12月31日，精度为3.33毫秒
Decimal	Decimal	固定精度和小数位数数值，在-10E38-1～10E38-1之间
Float	Double	-1.79E+308到1.79E+308范围内的浮点数
Image	Byte[]	二进制数据的可变长度流，范围在0到2E31-1字节之间
Int	Int32	32位的有符号整数
Money	Decimal	货币值，范围在-2E63～2E63-1之间，精度为千分之十个货币单位
NChar	String	Unicode字符的固定长度流，范围在1～4000个字符之间
NText	String	Unicode数据的可变长度流，最大长度为2E30-1个字符
NVarChar	String	Unicode字符的可变长度流，范围在1～4000个字符之间
Real	Single	-3.40E+38～3.40E+38范围内的浮点数
UniqueIdentifier	GUID	全局唯一标识符
SmallDateTime	DateTime	日期和时间数据，值范围从1900年1月1日到2079年6月6日，精度为1分钟
SmallInt	Int16	16位的有符号整数
SmallMoney	Decimal	货币值，范围在-214748.3648～+214748.3647之间，精度为千分之十个货币单位
Text	String	非Unicode数据的可变长度流，最大长度为2E31-1个字符
Timestamp	Byte[]	自动生成的二进制数，并保证其在数据库中唯一。存储大小为8字节。
TinyInt	Byte	8位的无符号整数
VarBinary	Byte[]	二进制数据的可变长度流，范围在1～8000个字节之间
VarChar	String	String。非Unicode字符的可变长度流，范围在1～8000个字符之间
Variant	Object	特殊数据类型，可以包含数值、字符串、二进制或日期数据，以及SQLServer值Empty和Null
Xml	XmlReader	XML值，使用GetValue方法或Value属性获取字符串形式的XML，或通过调用CreateReader方法获取XmlReader形式的XML
Udt		SQLServer2005用户定义的类型
Structured		指定表值参数中包含的构造数据的特殊数据类型
Date	DateTime	日期数据，值范围从公元1年1月1日到公元9999年12月31日
Time	DateTime	基于24小时制的时间数据。时间值范围从00:00:00～23:59:59.9999999，精度为100毫微秒
DateTime2	DateTime	日期和时间数据，日期值范围从公元1年1月1日到公元9999年12月31日。时间值范围从00:00:00到23:59:59.9999999，精度为100毫微秒
DateTimeOffset	DateTime	显示时区的日期和时间数据，日期值范围从公元1年1月1日到公元9999年12月31日。时间值范围从00:00:00～23:59:59.9999999，精度为100毫微秒。时区值范围从-14:00～+14:00

🔔注意：虽然表 12-7 列出了很多个 SqlDbType 枚举的值，实际上常用的主要有 3 大类：数值类、字符串类和日期时间类，其他的类型在开发及数据库设计中都很少使用。

12.3.2　管理 SqlParameter 对象集合

在 SqlCommand 在执行带参数的 SQL 命令之前，必须要为 SqlCommand 添加需要的 SqlParameter 对象。首先，通常是通过构造函数创建 SqlParameter 对象，并指定它的参数名、数据类型和值 3 个基本要素。然后，将该 SqlParameter 对象添加到 SqlCommand.Parameters 属性中。最后，通过 SqlCommand 执行命令，类似于 12.2.4 和 12.2.5 节介绍的方法。

SqlParameter 类的构造函数包括 7 个重载版本，不同的版本为 SqlParameter 指定不同的初始值，其中最常用的几个版本定义如下。

- SqlParameter()：默认构造函数，创建一个没有任何初始化的 SqlParameter 对象。在使用之前至少需要指定参数名称和参数值。
- SqlParameter(string parameterName, object value)：创建一个名称为 parameterName，且值为 value 的 SqlParameter 对象，根据 value 的类型自动判断参数的数据类型。
- SqlParameter(string parameterName, SqlDbType dbType)：创建一个名称为 parameterName，且类型为 dbType 的 SqlParameter 对象，使用之前需要指定参数的值。
- SqlParameter(string parameterName, SqlDbType dbType, int size)：创建一个名称为 parameterName、类型为 dbType，且数据最大为 size 字节的 SqlParameter 对象，使用之前需要制定参数的值。

创建 SqlParameter 对象之后，可以通过 SqlCommand.Parameters 属性对命令所需要的参数进行管理，通常是通过 Add() 添加参数。也可以用 foreach 关键字遍历 SqlCommand.Parameters 属性，来获取当前命令中已经添加的所有可用的参数，并对它的值进行修改。

如示例代码 12-4 演示创建并添加 SqlParameter 对象，首先，创建 SqlCommand 对象 cmd，并指定它的查询命令为一个包含两个参数@id 和@name 的 SELECT 命令；然后，分别通过 SqlParameter 两个版本的构造函数创建两个 SqlParameter 对象，指定它们的名称、类型、值等属性，并添加到 cmd.Parameters 中；最后，通过 foreach 关键字遍历 cmd.Parameters 属性，并打印出参数的信息。

示例代码 12-4

```
static void CreateParametersFunc( )
{
    //创建 SqlCommand 对象
    SqlCommand cmd = new SqlCommand( );
    //设置 SqlCommand 查询命令
    cmd.CommandText = "SELECT * FROM Users WHERE LoginID=@id OR Name=@name";
    //创建参数@id，并添加到 SqlCommand.Parameters 属性
    SqlParameter paraID = new SqlParameter( );
    paraID.ParameterName = "@id";
```

```
    paraID.DbType = System.Data.DbType.String;
    paraID.SqlDbType = System.Data.SqlDbType.NVarChar;
    paraID.Direction = System.Data.ParameterDirection.InputOutput;
    paraID.IsNullable = true;
    paraID.Value = "User1";
    cmd.Parameters.Add(paraID);
    //创建参数@name，并添加到 SqlCommand.Parameters 属性
    SqlParameter paraName = new SqlParameter("@name", System.Data.
    SqlDbType.NVarChar);
    paraName.DbType = System.Data.DbType.String;
    paraName.Value = "李四";
    cmd.Parameters.Add(paraName);
    //遍历所有 SqlCommand 中所有的 SqlParameter 对象，并打印提示信息
    foreach (SqlParameter para in cmd.Parameters)
    {
        System.Console.WriteLine("参数 [{0}] 的信息为:", para.ParameterName);
        System.Console.WriteLine("  DbType:     {0}", para.DbType);
        System.Console.WriteLine("  SqlDbType:  {0}", para.SqlDbType);
        System.Console.WriteLine("  Direction:    {0}", para.Direction);
        System.Console.WriteLine("  IsNullable:   {0}", para.IsNullable);
        System.Console.WriteLine("  Value:       {0}", para.Value);
    }
}
```

示例代码 12-4 的输出如下，从中可以看出 SqlParameter 默认为不允许为空（如参数@name），默认的 Direction 为输入参数（如参数@name）。

```
参数 [@id] 的信息为:
  DbType:     String
  SqlDbType:  NVarChar
  Direction:    InputOutput
  IsNullable:   true
  Value:      User1
参数 [@name] 的信息为:
  DbType:     String
  SqlDbType:  NVarChar
  Direction:    Input
  IsNullable:   false
  Value:      李四
```

技巧： 通过 SqlCommand.Parameters 属性的 Remove()、RemoveAt()方法可以删除指定的参数，通过 Clear()方法删除所有参数，通过 Insert()方法可以将参数添加到指定位置。

12.3.3　用 SqlParameter 传递数据

在 SQL 命令中，参数除了用来传入参数值之外，还可以作为查询的返回值，SqlParameter 同样具有这样的功能，如表 12-6 所示。它的 Direction 属性指定了参数的方向，Direction 属性是枚举类型 ParameterDirection，具有如下几个可选值。

- Input：表示该参数为输入参数，只能传入参数的值。
- Output：表示该参数只是输出参数，可以传出值，但是不能传入参数值。
- InputOutput：表示该参数既是输入参数，又是输出参数。

- ReturnValue：表示该参数是存储过程、内置函数或用户定义函数之类的操作的返回值。

在使用一个 SqlCommand 和 SqlParameter 执行带参数的 SQL 命令之前，对参数的 Direction 属性一定要设置正确，否则可能导致不能正常执行。如果参数是输入参数，且不允许为空，则使用前必须设置合法的 Value 属性，如果参数是输出参数，则可以在 SQL 命令执行完成后通过 Value 属性获取返回的值。

示例代码 12-5 演示如何使用输入参数传入数据。首先，创建一个数据库连接 connection，从 connection 创建一个 SqlCommand 对象 cmd，并打开数据库连接；然后，CommandText 属性设置要执行的 SQL 命令，该命令包含一个@minAge 参数，创建 SqlParameter 对象并添加到 cmd.Parameters 中；之后，通过 ExecuteReader()方法执行命令并获取 SqlDataReader 对象 reader；最后，通过 reader 对象的 GetString()、GetInt32()等方法获取并打印记录的值；最后，关闭数据库连接。

示例代码 12-5

```
static void ExecuteParameterSQL( )
{
    //连接字符串，使用数据库 UserLog
    string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
    Catalog=UserLog;Integrated Security= true ";
    //用连接字符串 conStr 创建 SqlConnection 对象 connection
    SqlConnection connection = new SqlConnection(conStr);
    //通过 SqlConnection 创建 SqlCommand 对象
    SqlCommand cmd = connection.CreateCommand( );
    connection.Open( );    //打开数据库连接
    //设置 SQL 命令
    cmd.CommandText = "SELECT Name, Age, XingBie, Mobile FROM Users WHERE
    Age>@minAge";
    //设置命令参数@minAge
    SqlParameter paraMinAge = new SqlParameter("@minAge",
    System.Data.SqlDbType.Int);
    paraMinAge.DbType = System.Data.DbType.Int32;
    paraMinAge.Value = 23;
    paraMinAge.Direction = System.Data.ParameterDirection.Input;
    cmd.Parameters.Add(paraMinAge);
    //执行 SQL 命令
    SqlDataReader reader = cmd.ExecuteReader( );
    System.Console.WriteLine("查询结果集如下：");
    while (reader.Read( ))         //重复读取全部读完
    {
        //读取并打印 Name，索引为 0，string 类型
        System.Console.Write("{0}\t", reader.GetString(0));
        //读取并打印 XingBie，索引为 2，string 类型
        System.Console.Write("{0}\t", reader.GetString(2));
        //读取并打印 Age，索引为 1，int 类型
        System.Console.Write("{0}\t", reader.GetInt32(1));
        //读取并打印 Mobile，索引为 3，string 类型
        System.Console.Write("{0}\t", reader.GetString(3));
        System.Console.WriteLine( );
    }
```

```
    //关闭数据库连接
    connection.Close( );
}
```

示例代码 12-5 的输出如下，与示例代码 12-3 的输出对比可以看出，由于这里的 SELECT 命令有一个 WHERE 子句，并且指定@minAge 的值为 23，所以只有年龄大于 23 的用户被查询出来。由此可见，SqlParameter 的确产生了期待的作用。

```
查询结果集如下：
李花     女      24      133331234577
```

注意：SqlCommand 在执行带有参数的 SQL 命令之前，在 Parameters 属性中必须要包含 SQL 命令所需的全部参数，而且不允许为空的参数必须具有可用的值，否则不能正确执行。

12.4　ADO.NET 无连接模式访问数据库

在 ADO.NET 有连接模式下，数据库操作都是实时的，数据处理逻辑通常时间较短，有时这样的实现不能满足复杂的处理逻辑。所以就需要用到 ADO.NET 无连接模式进行数据库访问，本节将介绍该技术。

12.4.1　了解 SqlDataAdapter 适配器类

在 ADO.NET 中，无连接模式访问数据库通常是将数据从数据库服务器通过 SQL 查询命令获取到内存中的 DataSet 或 DataTable 中，并且断开与数据库的连接。然后，在内存中根据业务逻辑对 DataSet 和 DataTable 中的数据进行任何合理的运算。最后，再连接到数据库，将 DataSet 和 DataTable 中的更改提交到数据库服务器。由此可见，无连接模式访问数据库具有如下优势：

- 对数据库连接的占用时间较短，因为只有需要进行交互时才连接到数据库，可以大大减轻数据库服务器的负担。
- 由于 DataSet 和 DataTable 是在内存中模拟的关系数据库，所以可以像操作数据库那样在内存中队数据进行处理，从而实现非常复杂的逻辑。
- 在对 DataSet 和 DataTable 进行处理时，可以利用 LINQ（第 14 章将介绍该内容）实现更加高效和复杂的查询操作。
- 在对 DataSet 和 DataTable 进行处理时，可以在内存中对更改数据进行验证，保证提交到数据库服务器的数据都是有效的。

在 ADO.NET 中，通过 SqlDataAdapter 和 DataSet 联合使用实现基于无连接的数据库访问。SqlDataAdatpter 类作为本地 DataSet（或 DataTable）与数据库服务器之间的连接器，它提供用于填充 DataSet 和更新 SQL Server 数据库的一组数据命令和一个数据库连接，表 12-8 列出了它的常用成员。

表 12-8　SqlDataAdapter类常用成员

分类	名　　称	访问性	说　　明
属性	AcceptChangesDuringFill	Public	读写属性，表示在任何Fill操作过程中，是否接受本地记录中已经存在的数据记录更改
	AcceptChangesDuringUpdate	Public	读写属性，表示在Update操作过程中，是否接受本地记录中已经存在的数据记录更改
	ContinueUpdateOnError	Public	读写属性，表示在行更新过程中遇到错误时是否继续更新下一条记录，还是产生异常（停止更新操作）
	UpdateBatchSize	Public	读写属性，表示每次到服务器的往返过程中处理的数据记录行数
	DeleteCommand	Public	读写属性，表示从数据库服务器删除记录所使用的Transact-SQL语句或存储过程
	InsertCommand	Public	读写属性，表示向数据库服务器添加记录所使用的Transact-SQL语句或存储过程
	SelectCommand	Public	读写属性，表示从数据库服务器获取记录所使用的Transact-SQL语句或存储过程
	UpdateCommand	Public	读写属性，表示更新数据库服务器记录所使用的Transact-SQL语句或存储过程
方法	Fill	Public	从数据库服务器获取数据，填充到本地的数据集（DataSet）或数据表（DataTable）
	FillSchema	Public	从数据库服务器获取数据架构，填充到本地的数据集（DataSet）或数据表（DataTable）
	Update	Public	将本地数据集或数据表中的数据记录更改更新到数据库服务器，为 DataSet 中每个已插入、已更新或已删除的行调用相应的INSERT、UPDATE或DELETE语句

从表 12-8 中可以看出，SqlDataAdapter 类的常用成员并不多，使用也非常简单。其中，Fill()方法用于从数据库服务器获取数据并填充到 DataSet 和 DataTable 中。Update()方法用于从将 DataSet 和 DataTable 中的数据更改提交到数据库服务器。

SqlDataAdapter 类的 Fill()和 Update()方法都是自动打开和关闭数据库连接。在执行之前至少要指定它的 SelectCommand 属性，该 SqlCommand 对象记录了 Fill()方法用来查询数据的 SQL 命令。Update()方法会根据数据的更改，自动调用 DeleteCommand、InsertCommand 和 UpdateCommand 中的某一个提交数据。

提示： DeleteCommand、InsertCommand 和 UpdateCommand 通常不需要明确指定，通过 SqlCommandBuilder 类可以根据 SqlDataAdapter 的 SelectCommand 自动生成对应的命令，只要查询数据中有一个是主键。

12.4.2　用 SqlDataAdapter 获取数据

本小节将介绍如何从数据库获取数据记录。使用 DataAdapter 类从数据库服务器获取数据记录到本地通常需要以下几个步骤：

（1）通过 DataAdapter 类构造函数创建一个可用的 DataAdapter 对象，同时为它指定数据库连接、查询命令等基本参数。DataAdapter 构造函数包括以下常用重载版本。

- SqlDataAdapter()：创建一个默认的 SqlDataAdapter 对象，它的任何参数都在后期指定。
- SqlDataAdapter(SqlCommand cmd)：创建一个具有指定查询命令的 SqlDataAdapter 对象，参数 cmd 表示用于从数据库获取数据的 SQL 命令。
- SqlDataAdapter(string selectcmd, SqlConnection con)：创建一个具有指定查询命令和数据库连接的 SqlDataAdapter 对象，其中 slectcmd 表示查询命令的 SQL 语句，con 表示可用的数据库连接。
- SqlDataAdapter(string selectcmd, string constr)：创建一个具有指定查询命令和数据库连接的 SqlDataAdapter 对象，其中 selectcmd 表示查询命令的 SQL 语句，constr 表示数据库连接的连接字符串。

（2）通过 SqlDataAdapter 类的 SelectCommand 属性设置或修改查询命令。同时自动产生 DeleteCommand、InsertCommand 和 UpdateCommand。

（3）通过 SqlDataAdapter 类的 FillScheme()方法从数据库服务器获取数据架构到本地数据集或数据表。如果只是需要数据结构，则必须这样。

（4）通过 SqlDataAdapter 类的 Fill()方法从数据库服务器获取数据到本地 DataSet 或 DataTable。Fill()方法包括以下常用重载版本。

- int Fill(DataSet dataSet)：根据 SQL 查询命令从数据库获取数据，并自动创建一个名为 Table 的 DataTable 对象来保存数据，然后将 DataTable 对象添加到 dataSet 中。
- int Fill(DataTable dataTable)：根据 SQL 查询命令从数据库获取数据，并自动将数据保存到 dataTable 对象中。

通常情况下，只需要使用 Fill(DataSet dataSet)这个简单版本即可。示例代码 12-6 演示如何使用 SqlDataAdapter 获取数据。FetchData()方法首先创建一个 SqlConnection 对象 connection 和 SqlCommand 对象 cmd，然后将 cmd 作为查询命令创建一个 SqlDataAdapter 对象 adapter；之后，创建 DataSet 对象 ds，并通过 SqlDataAdapter.Fill()方法从数据库获取数据并填充到 ds 中；最后，遍历并打印查询结果。

示例代码 12-6

```
class Program
{
    static void Main(string[] args)
    {
        FetchData( );
    }
    static void FetchData( )
    {
        //连接字符串，使用数据库 UserLog
        string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
        Catalog=UserLog;Integrated Security= true ";
        //用连接字符串 conStr 创建 SqlConnection 对象 connection
        SqlConnection connection = new SqlConnection(conStr);
        //通过 SqlConnection 创建 SqlCommand 对象
        SqlCommand cmd = connection.CreateCommand( );
        //设置 SQL 命令
        cmd.CommandText = "SELECT Name, Age, XingBie, Mobile FROM Users";
        SqlDataAdapter adapter = new SqlDataAdapter(cmd);
                                                   //创建 SqlDataAdapter 对象
```

```
        DataSet ds = new DataSet( );          //创建 DataSet 对象 ds
        adapter.Fill(ds);                     //从数据库获取数据，并填充到 ds 中
        //通过 DataSet.Tables 获取包括查询结果的 DataTable 对象 dt
        DataTable dt = ds.Tables["Table"];
        //遍历并打印查询结果
        System.Console.WriteLine("查询结果如下：");
        foreach (DataRow row in dt.Rows)
        {
            System.Console.Write("  {0}, ", row["Name"]);
            System.Console.Write("{0}, ", row["XingBie"]);
            System.Console.Write("{0}, ", row["Age"]);
            System.Console.Write("{0}", row["Mobile"]);
            System.Console.WriteLine( );
        }
    }
}
```

示例代码 12-6 的输出如下，从中可以看出 SqlDataAdapter 的确将数据获取并填入 DataSet 中。

```
查询结果如下：
  张三，男，23，131123456789
  李花，女，24，133331234577
```

技巧： 通过 SqlDataAdapter.Fill()获取数据之前，SqlDataAdapter 会自动打开数据库连接，在数据库获取完成后，自动关闭数据库连接，更加易于使用。

12.4.3　用 SqlDataAdapter 修改数据

要通过 SqlDataAdapter 修改数据，并将更改提交到数据库服务器，这就需要使用到 SqlDataAdapter 的 InsertCommand、DeleteCommand 和 UpdateCommand 这 3 个属性，它们分别表示插入记录、删除记录和更新记录时要调用的 SQL 命令。

值得庆幸的是，通常开发人员不需要明确为 SqlDataAdapter 指定 InsertCommand、DeleteCommand 和 UpdateCommand，可以通过 SqlCommandBuilder 类自动创建它们。SqlCommandBuilder 类可以根据 SqlDataAdapter 的 SelectCommand 命令自动生成用于更新数据的其他 3 个命令，这里主要用到它的 4 个方法，定义如下。

- public SqlCommandBuilder(SqlDataAdapter adapter)：使用指定的 SqlDataAdapter 对象创建一个 SqlCommandBuilder 对象，不用重新指定数据适配器。
- public SqlCommand GetDeleteCommand()：根据 SqlDataAdapter 自动生成，用于执行删除记录操作的 SQL 命令的 SqlCommand 对象。
- public SqlCommand GetInsertCommand()：根据 SqlDataAdapter 自动生成，用于执行插入记录操作的 SQL 命令的 SqlCommand 对象。
- public SqlCommand GetUpdateCommand()：根据 SqlDataAdapter 自动生成，用于执行更新记录操作的 SQL 命令的 SqlCommand 对象。

数据的更改是通过在内存中直接修改 DataSet 或 DataTable 来完成的，通过 SqlDataAdapter 主要是提交数据更改到数据库，通常需要以下几个步骤：

（1）通过 DataAdapter 类构造函数创建一个可用的 DataAdapter 对象，同时为它指定数据库连接、查询命令等基本参数。

（2）创建一个和 DataAdapter 类相关的 SQL 命令创建器（SqlCommondBuilder）对象，并通过 SqlCommandBuilder 类的 GetInsertCommand()、GetUpdateCommand()、GetDeleteCommand()方法获取与 DataAdapter 类的 Select 命令对应的删除、修改、添加命令。

（3）通过 DataAdapter 类的 FillScheme()或 Fill()方法，从数据库服务器获取数据记录到本地数据集 DataSet 或 DataTable 中。

（4）通过 DataSet 或 DataTable 的属性和方法等方式（见第 12 章）添加、删除、修改内存中的数据记录。

（5）通过 DataAdapter 类的 Update()方法将本地数据记录的修改提交到数据库服务器。其中 DataAdapter 类的 Update()方法具有多个重载版本，可以根据需要提交指定数据，它的常用重载定义如下。

- int Update(DataRow[] rows)：只提交指定行的数据到服务器，其中，参数 rows 表示要提交的数据记录的数组。
- int Update(DataSet ds)：提交指定数据集合中所有被更改的数据记录，其中，参数 ds 表示要提交的 DataSet。
- int Update(DataTable dt)：提交指定数据表中所有被更改的数据记录，其中，参数 dt 表示要提交的 DataTable。

示例代码 12-7 演示上面 5 个步骤的具体实例。UpdateData()方法首先创建一个 SqlConnection 对象 connection 和 SqlCommand 对象 cmd，将 cmd 作为查询命令创建一个 SqlDataAdapter 对象 adapter，并通过 SqlCommandBuilder 对象创建和设置 adapter 的 InsertCommand、UpdateCommand 和 DeleteCommand；然后，创建 DataSet 对象 ds，并通过 SqlDataAdapter.Fill()方法从数据库获取数据并填充到 ds 中；之后对 ds 中的记录进行修改，包括更改第 1 条记录的 Age 字段和添加 1 条新记录；最后，通过 SqlDataAdapter.Update()方法将更改提交到数据库服务器。

示例代码 12-7

```
class Program
{
    static void Main(string[] args)
    {
        UpdateData( );
        FetchData( );
    }
    static void FetchData( )
    {
        //此处省略代码，见示例代码 12-6
    }
    static void UpdateData( )
    {
        //连接字符串，使用数据库 UserLog
        string conStr = @"Data Source=WWW-818324B7DD9\YMYSQLSERVER;Initial
        Catalog=UserLog;Integrated Security= true ";
        //用连接字符串 conStr 创建 SqlConnection 对象 connection
        SqlConnection connection = new SqlConnection(conStr);
```

```
        //通过 SqlConnection 创建 SqlCommand 对象
        SqlCommand cmd = connection.CreateCommand( );
        //设置 SQL 命令
        cmd.CommandText = "SELECT LoginID, Password, Name, Age, XingBie,
        Mobile FROM Users";
        //创建 SqlDataAdapter 对象
        SqlDataAdapter adapter = new SqlDataAdapter(cmd);
        //创建 DataSet 对象 ds
        DataSet ds = new DataSet( );
        //创建 SqlCommandBuilder 对象
        SqlCommandBuilder cmdBuilder = new SqlCommandBuilder(adapter);
        //设置 InsertCommand、DeleteCommand、UpdateCommand。
        adapter.InsertCommand = cmdBuilder.GetInsertCommand( );
        adapter.DeleteCommand = cmdBuilder.GetDeleteCommand( );
        adapter.UpdateCommand = cmdBuilder.GetUpdateCommand( );
        //从数据库获取数据，并填充到 ds 中
        adapter.Fill(ds);
        //通过 DataSet.Tables 获取包括查询结果的 DataTable 对象 dt
        DataTable dt = ds.Tables["Table"];
        dt.Rows[0]["Age"] = 30;                    //更改第 1 条记录的 Age 字段
        dt.Rows.Add("User3", "User3Pwd", "杨阳", 26, "男", "13345678103");
        //添加 1 条新记录
        adapter.Update(ds);                        //将数据更改提交到数据库
    }
}
```

示例代码 12-7 的输出如下，与示例代码 12-6 的输出比较可以看出，第 1 条记录（名为“张三”）的年龄被改变为 30，而且新增了一条名为“杨阳”的记录。

```
查询结果如下：
  张三, 男, 30, 131123456789
  李花, 女, 24, 133331234577
  杨阳, 男, 26, 13345678103
```

注意： 上面的示例代码执行第 2 次时会发生异常，因为第 2 次执行插入记录操作时，LoginID 为 User3 的记录已经存在，会发生主键重复错误。另外，SelectCommand 中必须包含主键，才能自动创建 DeleteCommand、UpdateCommand、InsertCommand 等命令。

12.5　小　　结

ADO.NET 是一组向.NET 程序员公开数据访问服务的类，它们为创建分布式数据共享应用程序提供了一组丰富的组件，数据共享使用者应用程序可以使用 ADO.NET 连接到这些数据源，并检索、处理和更新所包含的数据。

ADO.NET 提供有连接和无连接两种数据库访问模式，并提供大量通用类完成数据库访问操作，这些包括数据库连接类（DbConnection）、SQL 命令类（DbCommand）、数据适配器类（DbDataAdapter）、数据读取器类（DbDataReader）等。

本章介绍 ADO.NET 数据库访问技术的基本概念和相关类的使用，通过本章的学习，

读者应该掌握以下知识点：

- 什么是 ADO.NET？
- ADO.NET 有哪些数据库访问模式？
- 如何通过 SqlConnection 类创建、打开、关闭数据库连接？
- 如何通过 SqlCommand 类创建和执行 SQL 命令？
- 如何通过 SqlDataReader 从数据库服务器读取数据？
- 如何通过 SqlParameter 类和 SqlCommand 类执行带有参数的 SQL 命令？
- 如何通过 SqlDataAdapter 类从数据库获取数据？
- 如何通过 SqlDataAdapter 类提交数据更改到数据库？

第 13 章　使用.NET 数据绑定

在很多用到数据库的软件中，从数据库查询数据、显示数据、对数据进行修改等都是基本的功能，开发人员通常需要专门为此开发大量的重复代码来实现这些功能。在.NET中，为了减轻这些工作量，提供了数据绑定技术。数据绑定是一种自动将数据按照指定格式显示到界面上的技术，本章将介绍.NET 数据绑定的基本概念和常用控件。

13.1　.NET 数据绑定基础

通过.NET 数据绑定，开发人员可以轻松地从数据库中获取数据，并将数据显示到界面，.NET 数据绑定可以绑定数据到 Windows Form 界面，也可以绑定到 ASP.NET 网页。本节将介绍.NET 数据绑定的基本概念，以及如何将数据绑定到 Windows Form 控件。

13.1.1　什么是数据绑定

很多软件都需要把数据从数据存储的地方（比如数据库、文件等）检索出来并呈现给用户，为了实现这一过程，开发人员不得不编写大量代码以图形的方式绘制这些数据，或者手工将这些数据赋值给控件属性来显示它。使用这种方式显示数据，对于不同类型的数据及不同类型的显示方式来说，处理方法都是不同的。有时还要允许用户通过用户界面修改或添加数据，这就需要编写更多代码从用户界面收集修改过的数据值，并将这些数据永久保存到数据原来存储的地方。这是一个相当复杂，而且容易出错的过程，于是需要一种比较通用的方式来完成这一过程。.NET 数据绑定就是其中一种很好的解决方案。

.NET 数据绑定将前面的过程分成多个可以独立操作的步骤，并将这些步骤封装进一些组件来帮助完成该过程。通过这些组件，开发人员可以避免开发大量的代码，从而大大提高开发效率。.NET 数据绑定还提供一些容易理解的模式，来规范编写代码的方法将数据挂接到可以显示和编辑数据的控件。另外，Visual Studio 2010 还提供了直观的设计时交互功能，可以大大提高数据绑定相关的开发和设计。

.NET 数据绑定主要包括 3 个主要的层次：数据显示控件、数据绑定管道和数据访问组件，它们分别完成各自的功能，如图 13-1 所示。

- 数据显示控件：这是一组界面元素，主要负责显示数据和接受用户的输入。包括 Windows Form 控件、ASP.NET 控件、WPF 控件等，在不同的技术背景进行不同的取舍。
- 数据绑定通道：该组件主要是 BindingSource 类，它是数据源和数据显示控件之间的纽带，将数据从数据源传递到显示控件，也从控件获取数据，并对数据进行必

要的处理。

- ❑ 数据访问组件：该组件负责从数据源获取数据，并将数据保存在内存中。数据源可以是任意类型的数据，比如数组、对象等，但是通常为数据库，这就需要 ADO.NET 组件。

.NET 数据绑定，可以将数据绑定到多种类型的控件，包括 ASP.NET 网页控件、Windows Form 控件、WPF 窗体控件等。本章重点介绍.NET 数据绑定与 Windows Form 控件。

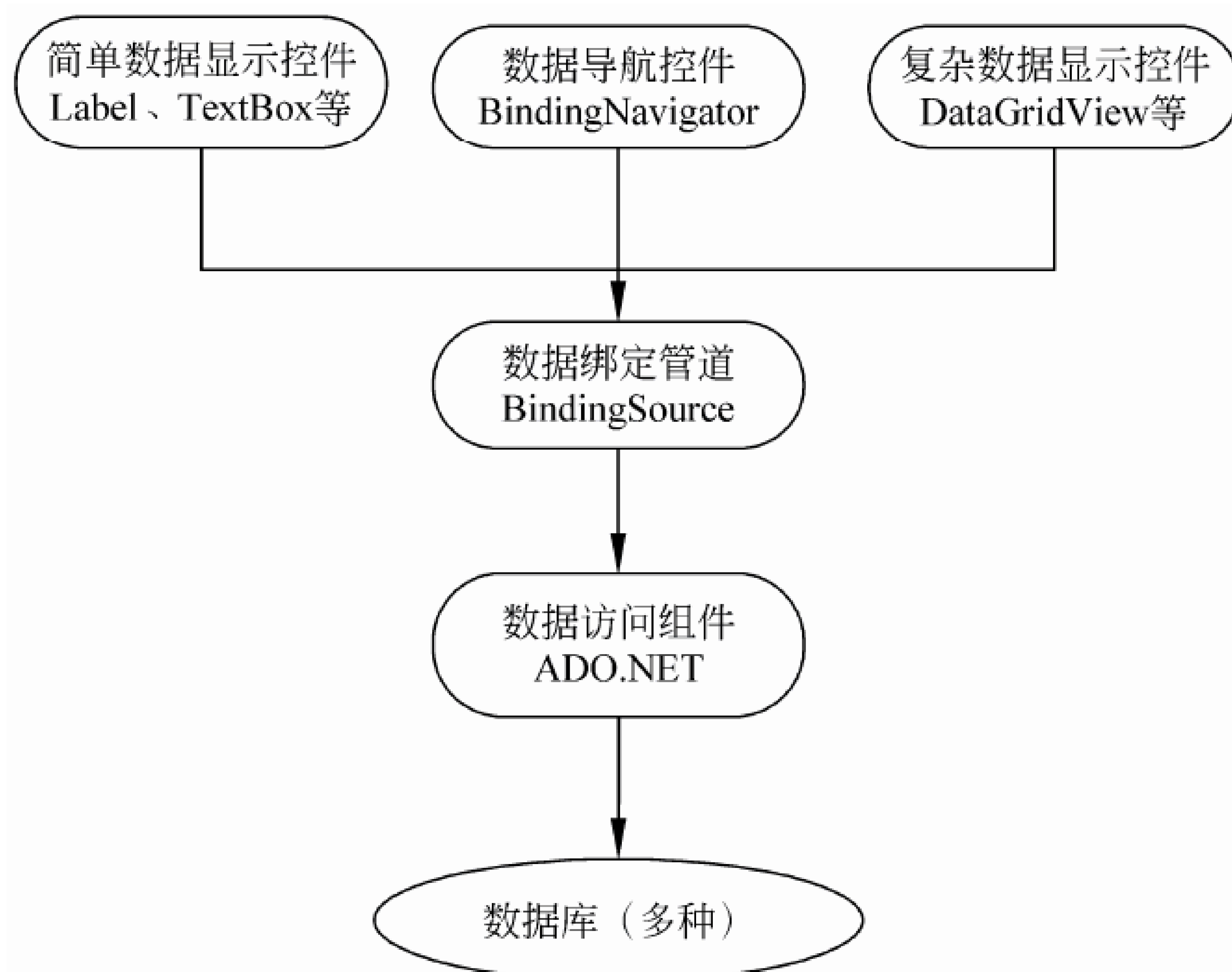


图 13-1　.NET 数据绑定结构图

13.1.2　了解 Windows 窗体数据绑定

在 Windows 窗体中，.NET 数据绑定不仅可以绑定到传统的数据源，还可以绑定到几乎所有包含数据的结构。可以绑定到值的数组，这些值可以在运行时计算、从文件中读取或者从其他控件的值派生。另外，还可以将任何控件的任何属性绑定到数据源。使用.NET 数据绑定不仅可以将控件的显示属性（例如 TextBox 控件的 Text 属性）绑定到数据源，还可以将数据绑定到控件的其他属性。比如，可以使用.NET 数据绑定来完成下列功能：

- ❑ 设置图像控件（Image）的图形。
- ❑ 设置一个或多个控件的背景色。
- ❑ 设置控件的大小等外观特性。

由此可见，通过.NET 数据绑定，可以设置窗体上任何控件的任何运行时可访问属性的值，也是一种通过后台数据自动更新界面显示的方法，不仅是控件的数据，也包括控件的外观。Windows 窗体可以利用两种类型的数据绑定：简单绑定和复杂绑定。

- ❑ 简单数据绑定：将一个控件绑定到单个数据元素（如数据集表的列中的值）。该绑定类型通常用于只显示单个值的控件（如 TextBox、Label 等）。
- ❑ 复杂数据绑定：将一个控件绑定到多个数据元素（如数据表中的多个记录）。复杂绑定又被称作基于列表的绑定。支持复杂绑定的控件通常都可以显示多个并列的数据，如 DataGridView、ListBox 和 ComboBox 等。

本章重点是介绍 Windows 窗体数据绑定，首先介绍简单数据绑定，然后介绍复杂数据绑定，并且结合 ADO.NET 介绍数据绑定中数据库相关的操作和技术。

13.2　创建简单数据绑定

简单数据绑定通常只是将单个数据绑定到 Label 这样只显示单个数据的控件。本节将介绍简单数据绑定技术，并且介绍如何对数据表中的多个数据记录进行导航。

13.2.1　用 BindingSource 绑定数据源

从图 13-1 中可以看出，在数据绑定中，BindingSource 组件是非常重要的核心部分。BindingSource 组件主要有两个用途：

- 提供一个将窗体上的控件绑定到数据的间接层。通过将 BindingSource 组件绑定到数据源，然后将窗体上的控件绑定到 BindingSource 组件来完成。界面与数据所有的进一步交互（包括导航、排序、筛选和更新）都是通过调用 BindingSource 组件完成。
- BindingSource 组件是强类型的数据源，可以保证数据的安全和有效。

由于并非窗体上所有的控件都会被绑定到某个数据源，所以 BindingSource 组件通常仅仅作为窗体上部分组件的数据源。在 Visual Studio 2010 中，可以通过 DataBindings 属性将 BindingSource 绑定到控件，该属性可在“属性管理器”视图中访问。

在.NET 中，BindingSource 组件通常可以绑定到两种数据源：一是简单数据源，包括对象的单个属性或 ArrayList 这样的基本集合；二是复杂数据源，如数据库表。在设计或运行时，通过将 BindingSource 组件的 DataSource 和 DataMember 属性分别设置为数据库和表，可以将该组件绑定到复杂数据源。

BindingSource 组件作为.NET 数据绑定的核心组件，它管理者所有数据绑定，因此实现了用于访问和排序数据的操作。BindingSource 组件由类 System.Windows.Forms.BindingSource 实现，表 13-1 列出了它作为数据源管理器提供的接口，包括支持遍历数据、编辑数据、排序数据等功能的成员。

表 13-1　BindingSource的主要成员

分类	名　　称	说　　明
属性	AllowEdit	只读属性，表示是否可以编辑基础列表中的项
	AllowRemove	只读属性，表示是否可从基础列表中移除项
	AllowNew	读写属性，表示是否可以使用AddNew()方法向列表中添加项
	Count	只读属性，获取绑定数据中数据的总项数
	DataSource	读写属性，获取或设置连接器绑定到的数据源
	DataMember	读写属性，获取或设置当前绑定到的数据源中的特定列表
	List	只读属性，获取连接器绑定到的列表（数据源）
	Current	只读属性，获取数据源中的当前项

续表

分类	名　　称	说　　明
属性	Position	读写属性，获取或设置绑定列表中当前项的索引
	Item	读写属性，获取或设置数据源中指定索引处的元素
	IsFixedSize	只读属性，表示绑定的列表是否具有固定大小
	IsReadOnly	只读属性，表示绑定的列表是否为只读
	IsSorted	只读属性，表示是否可以对绑定列表中的项排序
	Sort	读写属性，表示用于排序的列名称及用于查看数据源中的行的排序顺序
	SortDescriptions	只读属性，表示应用于数据源排序说明的集合
	SortDirection	只读属性，表示列表中项的排序方向
	SupportsSorting	只读属性，表示数据源是否支持排序
	SupportsFiltering	只读属性，表示数据源是否支持筛选
	SupportsSearching	只读属性，表示数据源是否支持使用 Find() 方法进行搜索
方法	Add	将现有项添加到数据源中
	Insert	将指定元素插入到数据源中指定的索引处
	AddNew	向数据源中添加新项
	Remove	从数据源中移除指定的元素
	RemoveAt	从数据源中移除指定索引处的元素
	RemoveCurrent	从数据源中移除当前元素
	Clear	从数据源中移除所有元素
	CancelEdit	取消当前的编辑操作，被编辑的数据会恢复到编辑前的数据
	EndEdit	将未提交的更改应用于数据源，提交之后不能通过CancelEdit()来取消
	GetListName	获取为绑定提供数据的列表（即数据源）名称
	MoveFirst	将当前元素指针移至数据源中的第一个元素
	MoveLast	将当前元素指针移至数据源中的最后一个元素
	MoveNext	将当前元素指针移至数据源中的下一个元素
	MovePrevious	将当前元素指针移至数据源中的上一个元素
	ResetBindings	使绑定到数据源的控件重新读取列表中的所有元素，并刷新这些元素的显示值
	ResetCurrentItem	使绑定到数据源的控件重新读取当前选定的元素，并刷新其显示值
	ResetItem	使绑定到数据源的控件重新读取指定索引处的元素，并刷新其显示值
事件	AddingNew	在将新元素添加到数据源之前发生该事件
	BindingComplete	当所有被绑定的控件都已绑定到数据源时发生该事件
	CurrentChanged	在当前绑定元素指针发生更改时发生该事件
	CurrentItemChanged	在Current属性的属性值更改后发生该事件
	DataError	当绑定数据发生异常，而且由BindingSource无提示处理时发生该事件
	DataMemberChanged	在DataMember属性值更改后发生该事件
	DataSourceChanged	在DataSource属性值更改后发生该事件
	ListChanged	当数据源更改或数据源中的项更改时发生该事件
	PositionChanged	在Position属性的值更改后发生该事件

从表 13-1 中可以看出，通过 BindingSource 类可以对数据绑定的数据源进行全面的控制，可以执行添加、删除、修改、取消修改等操作。也可以看出 BindingSource 背后其实有两种可能的数据源，分别包含在 DataMember 和 DataSource 中，通常在通过 Visual Studio 2010 的向导进行数据绑定的配置时，它会自动选择合适的成员进行赋值。

另外，通过 BindingSource 类提供的事件，可以监视数据绑定当前正在执行的动作，通过响应这些事件可以进行特定的处理，比如，在数据绑定过程中需要让界面处于不可用状态，就可以在启动时将界面设置为不可用（Disabled），在 BindingComplete 事件中再将界面设置为可用（Enabled）即可。

虽然 BindingSource 类是数据绑定的核心，是绑定控件和数据源之间的纽带，但是它本身是一个后台辅助类，通常不会直接使用。所以，本节只是介绍它的基本概念和功能，通过了解这些内容读者可以感觉到数据绑定其实并不神秘。本章后面几节会介绍数据绑定的具体实例。

13.2.2　用 BindingNavigator 进行导航

在很多与数据相关的软件中，数据导航是常见的功能，所以 Windows Form 控件库提供了控件 BindingNavigator。它本质上是一个包含多个内置工具栏项的工具栏控件，如图 13-2 所示，通过 BindingSource 属性指定与它协同工作的数据绑定。

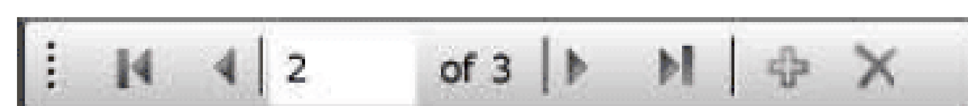


图 13-2　BindingNavigator 控件

从图 13-2 中可以看出，BindingNavigator 控件一共包括 6 个按钮、1 个输入框、1 个 Label 控件，它们分别具有不同的图标和功能，集成到一起完成数据导航的功能。具体如下：

- ❑ MoveFirstItem 按钮：执行 BindingSource 的 MoveFirst()方法，将当前元素指针移到数据源的第一个元素。
- ❑ MoveLastItem 按钮：执行 BindingSource 的 MoveLast()方法，将当前元素指针移到数据源的最后一个元素。
- ❑ MoveNextItem 按钮：执行 BindingSource 的 MoveNext()方法，将当前元素指针移到数据源中的下一个元素。
- ❑ MovePreviousItem 按钮：执行 BindingSource 的 MovePrevious()方法，将当前元素指针移到数据源中的上一个元素。
- ❑ AddNewItem 按钮：执行 BindingSource 的 AddNew()方法，向数据源中添加一个新的元素。
- ❑ DeleteItem 按钮：执行 BindingSource 的 RemoveCurrent()方法，从数据源中移除当前选中的元素。
- ❑ PositionItem 文本框 2 ：显示 BindingSource 中当前选中的元素索引，也可以输入并导航到指定索引的元素。
- ❑ CountItem 标签 of 3 ：显示 BindingSource 中元素的总数。

BindingNavigator 控件的通常与 BindingSource 成对出现，将 BindingSource 对象绑定到 BindingNavigator 控件的 BindingSource 属性即可。通常，通过 Visual Studio 2010 创建和使

用 BindingNavigator 控件更加方便，本节通过以下步骤创建一个使用该控件的实例。

（1）打开 Visual Studio 2010，并新建一个名为 UseBindingNavigator 的 Windows 窗体应用程序，如图 13-3 所示。



图 13-3　创建 UseBindingNavigator 项目

（2）从工具栏的“数据”栏找到“BindingSource”控件 BindingSource，并将其拖放到窗体设计器中。在“属性管理器”中将其命名为 bsUserLogs。

（3）此时 bsUserLogs 的数据源 DataSource 属性为空，在其下拉列表框中选择“添加项目数据源”操作启动数据源配置向导，为应用程序添加需要绑定的数据源。

（4）在数据源配置向导中，选择前面章节创建的数据库 UserLog 的表 Users 作为数据源，并命名为 UsersDataSet，如图 13-4 所示。单击“完成”按钮完成此操作。



图 13-4　选择 UserLog.Users 作为数据源

（5）此时，完成了 BindingSource 的设置。在属性管理器中查看 bsUserLogs，可以看

到它的 DataSource 属性被自动设置为第 4 步新建的数据源 UsersDataSet。这里还需要设置它的 DataMember 属性为 Users 表，如图 13-5 所示。

（6）从“工具箱”的“数据”栏中选择 BindingNavigator 控件 BindingNavigator，并将它拖放到窗体设计器中，在属性管理器中将其命名为 userNavigator。

（7）在属性管理器中设置 userNavigator 的 BindingSource 属性为前面创建的 bsUserLogs。

到此，运行应用程序的开发基本完成，运行将得到如图 13-6 所示的效果图。可以通过导航菜单导航数据库中的数据，在 13.2.3 节将看到更好的效果。

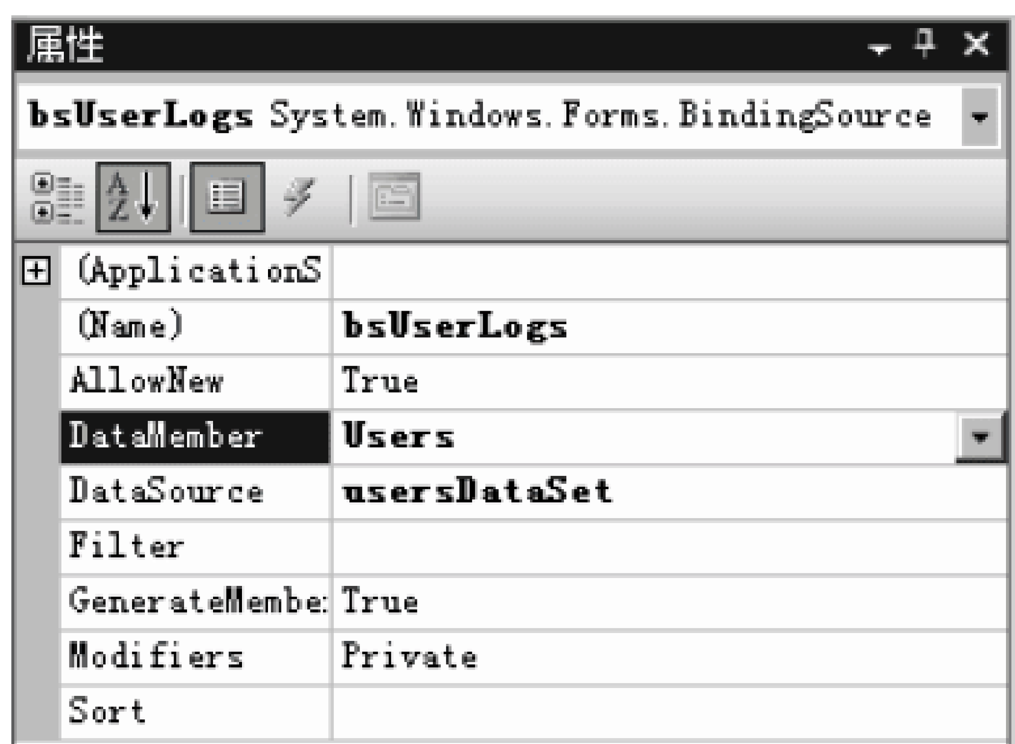


图 13-5　bsUserLogs 的属性

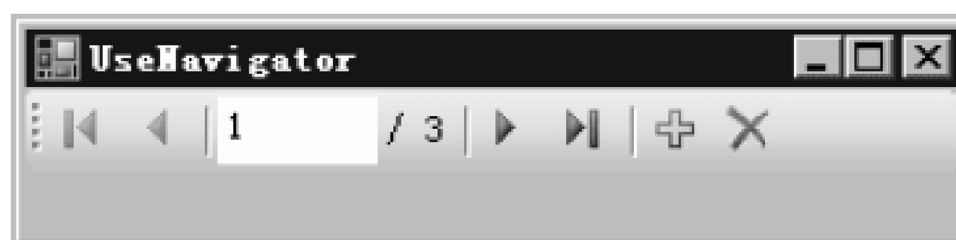


图 13-6　UseBindingNavigator 实例

在本例中，BindingNavigator（userNavigator）控件通过 BindingSource（bsUserLogs）控件从数据源 usersDataSet 的表 Users 中获取数据。并通过内置的导航功能对 Users 中的记录进行遍历。

技巧：如果在创建数据源失败时，可以先检查一下应用程序所在路径是否具有#、&等特殊字符，如果有可以修改路径进行解决。另外，可以通过“数据源”创建和查看数据源信息。

13.2.3　绑定数据到 TextBox 等控件

在 13.2.2 节创建的实例 UseBindingNavigator 中，导航控件虽然能够导航数据，但是并不能直观地看到导航效果，本节将数据绑定到 TextBox、Label 等简单控件，从而使得 UseBindingNavigator 实例更加清楚。

在.NET 中，可以通过数据绑定将数据绑定并显示到 TextBox、Label 等普通控件中，这点通常是在“属性管理器”中通过 DataBindings 属性来进行配置，但是不同控件可以绑定的数据类型不同。如图 13-7 所示为 TextBox 控件的 DataBindings 属性，由此可见，可以对 TextBox 控件的 Text 属性和 Tag 属性进行绑定。Tag 属性通常用于控件所表示的后台数据，而 Text 属性则表示要显示的数据。

为了进一步完善 UseBindingNavigator 实例，需要将数据表 Users 中需要显示的列绑定到各个控件中，大致需要如下几步。

（1）打开实例 UseBindingNavigator，并通过“数据”｜“显示数据源”菜单打开“数据源”视图，从中打开 UserDataSet.Users 表，如图 13-8 所示，查看 Users 中有哪些列。

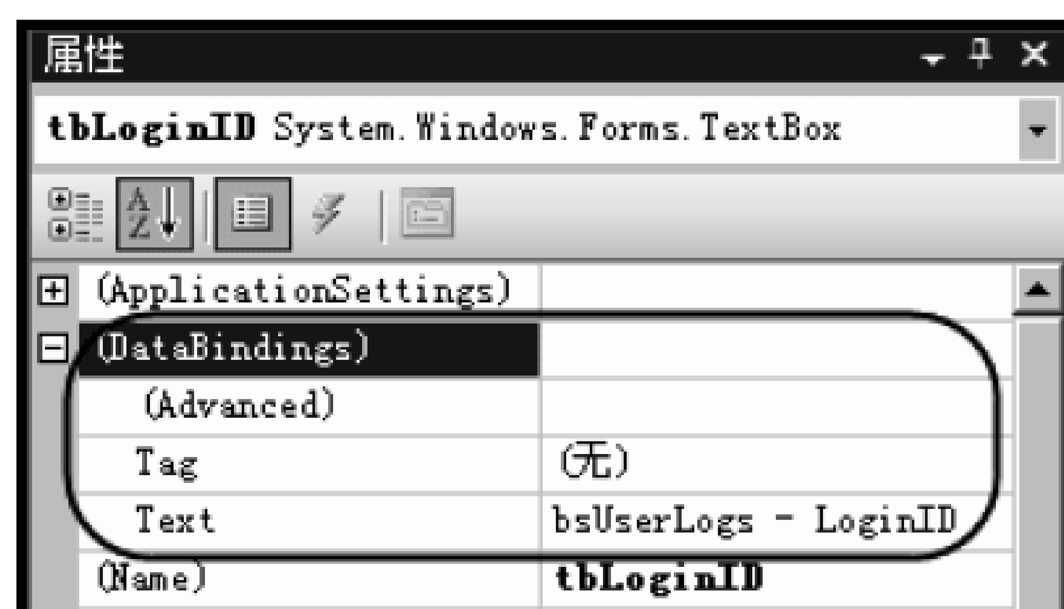


图 13-7　DataBindings 结点

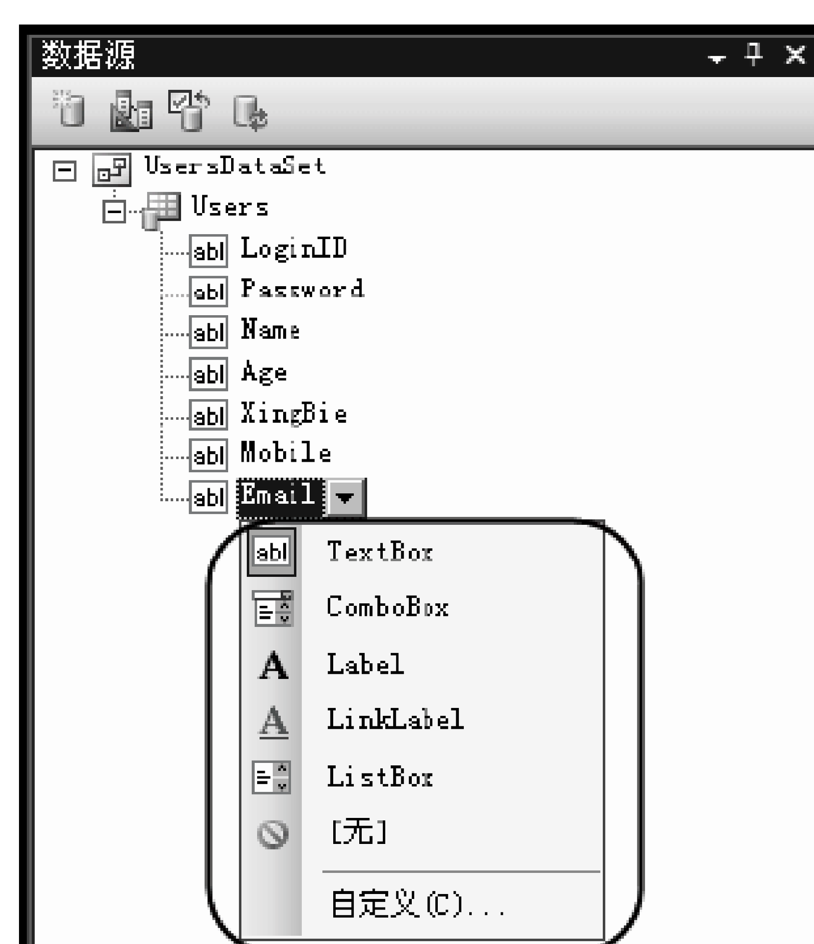


图 13-8　数据源视图

（2）根据“数据源”视图，为 Users 表的各个字段添加 1 个 Label 控件和 1 个 Text 控件，其中，Label 控件用来表示要显示的字段的名称，Text 控件用来表示字段的值。通过图 13-7 所示的方法，将 TextBox 的 Text 属性绑定到 Users 表的一个列，具体绑定信息如表 13-2 所示。

表 13-2　Users表的绑定信息

Label的名称	TextBox的名称	绑定到Users的字段
登录ID	tbLoginID	LoginID
密码	tbPassword	Password
姓名	tbName	Name
年龄	tbAge	Age
性别	tbXB	XingBie
电话	tbTel	Mobile
邮箱	tbEmail	Email

（3）对第 2 步产生的多个 Label 和 TextBox 控件进行合理布局，使其更加美观。

生成并新的 UseBindingNavigator 实例，可以看到如图 13-9 所示的运行效果，当通过导航控件切换到其他记录时，文本框中所有的字段信息都会随之更新，这就是数据绑定的方便之处。

在整个应用程序的开发过程中，开发人员不需要编写任何一行代码，只要拖放控件，并对控件进行必须的配置，就可以轻松完成数据库访问、导航数据、显示数据，甚至包括数据编辑等功能。由此可见，通过数据绑定，开发效率将得到大大提高。



图 13-9　UseBindingNavigator 实例

13.3　创建复杂数据绑定

除了前面介绍的数据导航控件外，.NET 类库还提供了一种表以格形式显示和编辑数据

的控件——DataGridView 控件，该控件既可以显示内存数据，也可以从数据绑定获取数据。本节详细介绍 DataGridView 控件的具体使用。

13.3.1　了解 DataGridView 控件

在.NET 4.0 中，以表格形式存储的数据通常是通过 DataGridView 控件来显示和编辑。DataGridView 控件替代了早期.NET 类库中的 DataGrid 控件，使用 DataGridView 控件，可以显示和编辑来自多种不同类型数据源的表格数据，包括以下几种数据类型的实例：

- 实现 IList 接口的类，这些类提供一维数组格式的数据，如 List，Array 等。
- 实现 IListSource 接口的类，这些类提供表格形式的数据，如 DataTable 类和 DataSet 类。
- 实现 IBindingList 接口的类，这些类提供可用于绑定的一维数组简单数据，如泛型类 BindingList<T>。
- 实现 IBindingListView 接口的类，这些类提供可用于绑定的复杂数据源，如 BindingSource 类。

DataGridView 控件既可以工作在绑定模式下，也可以在非数据绑定模式下工作。在非数据绑定模式下，开发人员可以往 DataGridView 控件添加数据，并控制数据的显示。所以，当需要显示的数据量很小，而且数据通常为运行时产生的临时数据时，可以在非数据绑定模式下使用 DataGridView。

在数据绑定模式下，只需要为 DataGridView 控件指定绑定的数据源，它可以自动从数据获取数据，并自动显示数据。所以，当需要显示或更新外部数据源的数据，而且数据量较大时，通常在数据绑定模式下使用 DataGridView，这也是 DataGridView 最常见的使用方式。

DataGridView 控件定义在 System.Windows.Forms 命名空间下，它提供了大量用于方便表格显示和操作的属性和方法，同时也提供了便于进行数据绑定的成员和方法。其中，DataGridView 主要的成员如表 13-3 所示。

表 13-3　DataGridView的主要成员

分类	名　称	说　明
属性	AllowUserToAddRows	读写属性，表示是否向用户显示添加行
	AllowUserToDeleteRows	读写属性，表示是否允许用户从DataGridView中删除行
	AllowUserToOrderColumns	读写属性，表示是否允许通过手动对列重新定位
	AllowUserToResizeColumns	读写属性，表示用户是否可以调整列的宽度
	AllowUserToResizeRows	读写属性，表示用户是否可以调整行的高度
	AutoGenerateColumns	读写属性，表示在设置DataSource或DataMember属性时是否自动创建列
	ReadOnly	只读属性，表示用户是否可以编辑DataGridView的单元格
	Columns	只读属性，表示DataGridView中的所有列的集合
	ColumnCount	读写属性，表示DataGridView中显示的列数
	SelectedColumns	只读属性，表示DataGridView中用户选定的列的集合
	SortedColumn	只读属性，表示DataGridView中当前排序所依据的列
	Rows	只读属性，表示DataGridView中的所有行的集合

续表

分类	名　　称	说　　明
属性	RowCount	读写属性，表示DataGridView中显示的行数
	SelectedRows	只读属性，表示DataGridView中用户选定的行的集合
	NewRowIndex	只读属性，表示新记录所在行的索引
	CurrentRow	只读属性，获取包含当前单元格的行
	IsCurrentRowDirty	只读属性，表示当前行是否有未提交的更改
	CurrentCell	读写属性，表示当前处于活动状态的单元格
	CurrentCellAddress	只读属性，表示当前活动单元格的行索引和列索引
	IsCurrentCellDirty	只读属性，表示当前单元格是否有未提交的更改
	IsCurrentCellInEditMode	只读属性，表示是否正在编辑当前活动单元格
	FirstDisplayedCell	读写属性，表示当前显示在DataGridView中的第一个单元格，此单元格通常位于左上角
	TopLeftHeaderCell	读写属性，表示位于DataGridView控件左上角的标题单元格
	SelectedCells	只读属性，表示用户选定的单元格的集合
	DataMember	读写属性，表示数据源中DataGridView显示其数据的列表或表的名称
	DataSource	读写属性，表示DataGridView所显示数据的数据源
	EditMode	读写属性，表示如何开始编辑单元格
	SelectionMode	读写属性，表示如何选择DataGridView的单元格
	MultiSelect	读写属性，表示是否允许一次选择多个单元格、行或列
	ShowCellErrors	读写属性，表示是否显示单元格错误
	SortOrder	只读属性，表示是否进行排序
	StandardTab	读写属性，表示按Tab键是否会将焦点按Tab键顺序移到下一个控件，而不是将焦点移到控件中的下一个单元格
方法	AreAllCellsSelected	返回一个值，表示当前是否选择了所有的单元格
	SelectAll	选择DataGridView中的所有单元格
	ClearSelection	取消对当前选定的单元格的选择
	AutoResizeColumn	自动调整指定列的宽度以适应其单元格的内容
	AutoResizeColumnHeadersHeight	自动调整列标题的高度以适应标题内容
	AutoResizeColumns	自动调整所有列的宽度以适应其单元格的内容
	AutoResizeRow	自动调整指定行的高度以适应其单元格的内容
	AutoResizeRowHeadersWidth	自动调整行标题的宽度以适应标题内容
	AutoResizeRows	自动调整某些或所有行的高度以适应其内容
	BeginEdit	将当前的单元格置于编辑模式下，用户可以编辑该单元格
	CommitEdit	将当前单元格中的更改提交到数据缓存，但不结束编辑模式
	EndEdit	提交对当前单元格进行的编辑并结束编辑操作
	CancelEdit	取消当前选定单元格的编辑模式并丢弃所有更改
	DisplayedColumnCount	返回向用户显示的列数
	DisplayedRowCount	返回向用户显示的行数
	GetCellCount	获取满足所提供筛选器的单元格的数目
	Sort	对DataGridView控件的数据进行排序
	UpdateCellErrorText	强制指定位置的单元格更新其错误文本
	UpdateRowErrorText	强制一行或多行更新其错误文本

从表 13-3 中可以看出，DataGridView 控件具有非常强大的表格数据编辑功能。此外，DataGridView 还提供了非常丰富的外观配置功能，在 13.3.2 节将介绍 DataGridView 控件的外观结构。

13.3.2　了解 DataGridView 控件外观

DataGridView 控件作为典型的表格数据显示和编辑控件，它的典型样式如图 13-10 所示，从中可以看出，DataGridView 将数据按照表格的形式显示，主要包括表（Grid）、行（Row）、列（Column）、单元格（Cell）4 个层次。

DataGridView 控件表示的表类似于数据库中的表，它包含多个列定义，每一列定义了数据的类型、显示和编辑模式等。同时也包含任意行数据，每行数据列定义中的数据。真正显示和编辑数据是通过单元格来实现的。

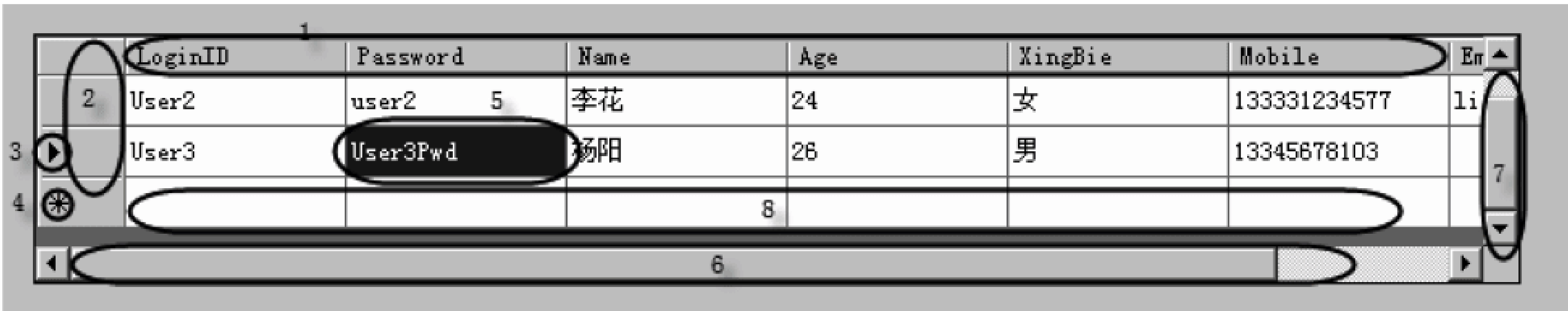


图 13-10　DataGridView 控件典型样式

在图 13-10 中，用数字对 DataGridView 控件的主要部分进行编号，主要如下：

- ❑ 列标题（编号 1）：用于显示表格的列标题，可以独立指定它的显示样式。
- ❑ 行标题（编号 2）：通常用于显示表格的行状态，包括错误、新建、选中等，同样可以独立指定它的显示样式。
- ❑ 选中标记（编号 3）：选中状态，表示该行为选中，且为当前行。
- ❑ 新行标记（编号 4）：新行标记，表示该行为新创建的行，而且存在未提交的更改。
- ❑ 选中单元格（编号 5）：选中单元格默认为蓝色背景，同样可以独立设置选中单元格的样式。
- ❑ 水平滚动条（编号 6）：用于水平滚动表格，可以指定是否需要启用该滚动条。
- ❑ 垂直滚动条（编号 7）：用于垂直滚动表格，可以指定是否需要启动该滚动条。
- ❑ 新建行（编号 8）：整个一行为新建行，如果某列有默认值，那么新建时会产生默认值，否则为空白，用户可以在该行的单元格中输入新数据。

除了上面介绍的外观特点以外，DataGridView 控件还具有很多其他的外观特点，但这些外观的设置并不是杂乱无章的，它通过样式（Style）将分散的外观特点很好地汇总到一起，并应用到 DataGridView 中各个单元上。表 13-4 列出了 DataGridView 中与外观样式相关的属性。

表 13-4　DataGridView的外观属性

名　　称	说　　明
AdjustedTopLeftHeaderBorderStyle	只读属性，表示DataGridView左上角单元格的边框样式
AdvancedCellBorderStyle	只读属性，表示DataGridView中单元格的边框样式
AdvancedColumnHeadersBorderStyle	只读属性，表示DataGridView中列标题单元格的边框样式

续表

名　　称	说　　明
AdvancedRowHeadersBorderStyle	只读属性，表示DataGridView中行标题单元格的边框样式
AlternatingRowsDefaultCellStyle	读写属性，表示DataGridView中奇数行的默认单元格样式
AutoSizeColumnsMode	读写属性，表示如何确定列宽
AutoSizeRowsMode	读写属性，表示如何确定行高
BackgroundColor	读写属性，表示DataGridView的背景色
BackgroundImage	读写属性，表示DataGridView控件的背景图片
BackgroundImageLayout	读写属性，表示DataGridView控件背景图像布局
BorderStyle	读写属性，表示DataGridView的边框样式
CellBorderStyle	只读属性，表示DataGridView的单元格边框样式
ClipboardCopyMode	读写属性，表示用户是否可以将单元格的文本值复制到剪贴板，以及是否包括行标题和列标题文本
ColumnHeadersBorderStyle	只读属性，表示应用于列标题的边框样式
ColumnHeadersDefaultCellStyle	读写属性，表示默认列标题样式
ColumnHeadersHeight	读写属性，表示列标题行的高度，以像素为单位
ColumnHeadersHeightSizeMode	读写属性，表示是否可以调整列标题的高度，以及它是由用户调整还是根据标题的内容自动调整
ColumnHeadersVisible	读写属性，表示是否显示列标题行
DefaultCellStyle	读写属性，表示DataGridView中默认的单元格样式
GridColor	读写属性，表示DataGridView中网格线的颜色
RowHeadersBorderStyle	读写属性，表示行标题单元格的边框样式
RowHeadersDefaultCellStyle	读写属性，表示应用于行标题单元格的默认样式
RowHeadersVisible	读写属性，表示是否显示包含行标题的列
RowHeadersWidth	读写属性，表示包含行标题的列的宽度，以像素为单位
RowHeadersWidthSizeMode	读写属性，表示是否可以调整行标题的宽度，以及它是由用户调整还是根据标题的内容自动调整
RowsDefaultCellStyle	读写属性，表示应用于DataGridView的行单元格的默认样式
ScrollBars	读写属性，表示在DataGridView控件中显示的滚动条的类型
ShowCellToolTips	读写属性，表示当鼠标指针停留在单元格上时，是否显示工具提示
ShowEditingIcon	读写属性，表示编辑标志符号是否在所编辑的单元格的行标题中可见
ShowRowErrors	读写属性，表示行标题是否为包含数据输入错误的每一行显示错误标志符号

从表 13-4 中可以看出，DataGridView 控件的外观相关的属性数量很多，要设置一个很漂亮的外观，不仅需要大量的属性设置工作，而且还需要比较好的美术功底。如图 13-11 为一个简单的 DataGridView 样式设计效果。

图 13-11　DataGridView 外观示例效果

13.3.3　编辑 DataGridView 的列信息

DataGridView 控件作为典型的以表格形式显示数据的控件，很多时候可以在非数据绑定模式下使用，用于显示和编辑少量数据。非绑定模式下使用 DataGridView 控件，必须设置它的列信息，大致通过以下步骤来设置列信息：

（1）打开 Visual Studio 2010，创建一个新的 Windows 窗体应用程序，并命名为 UnboundDataGridView。重命名窗体 Form1 为 FrmMain，并设置它的标题。

（2）从“工具箱”上拖动一个 DataGridView 控件到窗体 FrmMain 上，并命名为 dgvBooks。

（3）在窗体上选择控件 dgvBooks，通过右键弹出菜单“编辑列”或“添加列”命令为 dgvBooks 添加列信息，如图 13-12 所示。左边列表给出目前所有的列信息，右边属性编辑器可以用于设置指定列的类型、名称等信息。

（4）通过图 13-12 所示的“添加”按钮添加新的列信息，如图 13-13 所示。从图中可以看出，在添加一个列时，需要指定它是否可见、是否只读、是否冻结等属性。DataGridView 控件中的列有 3 个最重要的属性。

- 名称（Name）：在整个窗体中唯一标识该列，通过它可以访问该列并修改列的属性。
- 类型（Type）：表示该列在显示和编辑使用的控件类型，包括文本框（TextBox）、下拉框（CommboBox）、选择框（CheckBox）、按钮（Button）、图片（Image）和链接（Link）。
- 页眉文本（Caption）：即列标题，通常是可读性较好的文字。



图 13-12　编辑 DataGridView 列信息

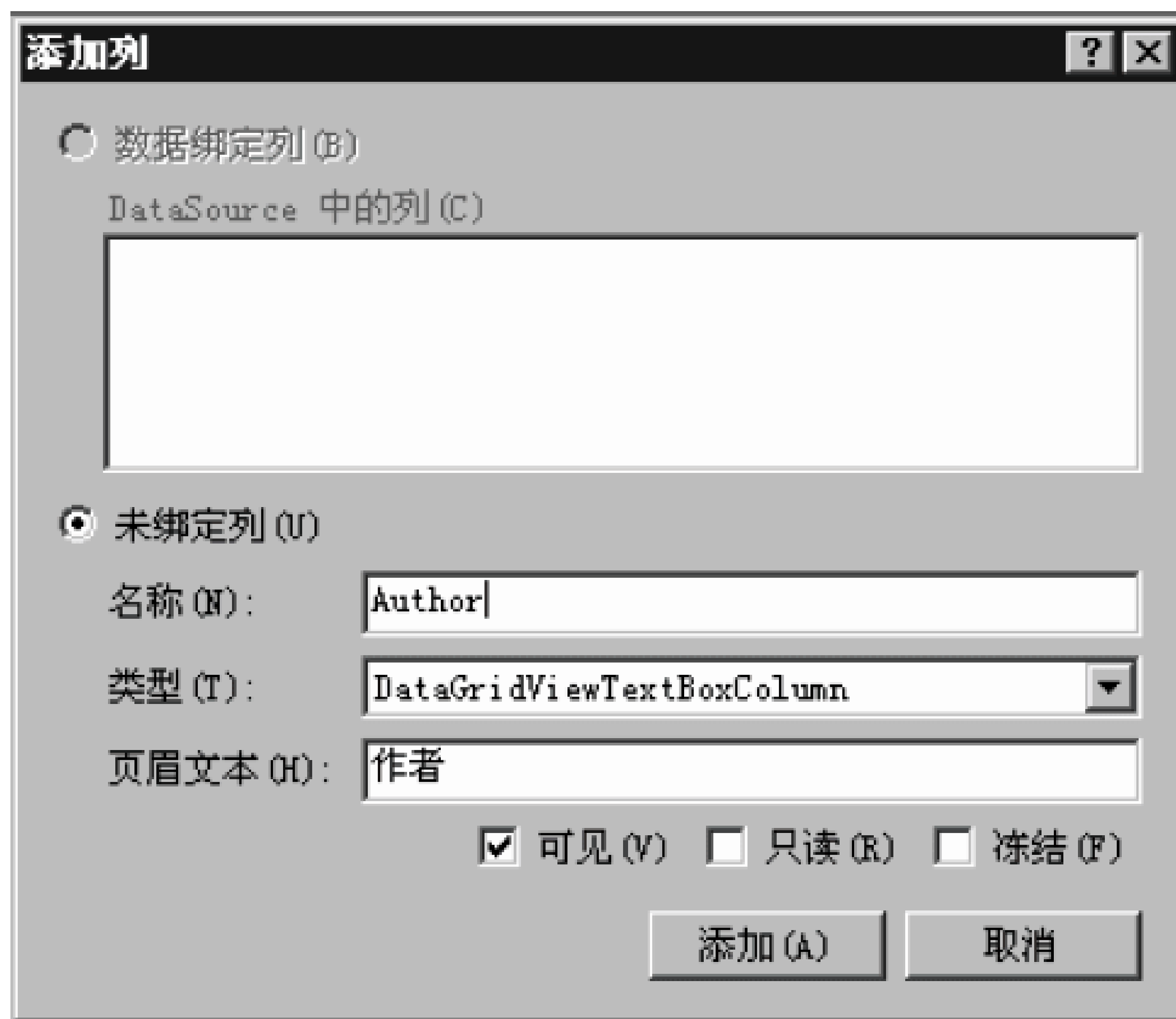


图 13-13　添加新列对话框

注意：在冻结和只读情况下，该列的字段都不能被编辑，但是冻结的列固定显示在表格的左边，不会因为滚动条的滚动而被隐藏。不可见的情况下，该列不会显示。

在实例 UnboundDataGridView 中，要显示和编辑一个图书记录的列表，所以添加如表 13-5 所示的列信息，其中“编号”和“书名”两列冻结。

表 13-5　DataGridView的外观属性

名　　称	标　　题	类　　型	是 否 冻 结
colBookID	编号	文本框	冻结
colBookName	书名	文本框	冻结
colAuthor	作者	文本框	不冻结
colPrice	价格	文本框	不冻结
colDate	日期	文本框	不冻结

技巧： 在 DataGridView 中，可以通过 Columns 属性访问所有列的集合，也可以通过列名称访问特定的列。

13.3.4　访问 DataGridView 的数据

到此为止，13.3.3 节的实例完成 DataGridView 控件列定义。通常，在非绑定下使用，还需要为每一行数据设计一个类，用来保存每一行的数据。本例中，创建一个 Book 类，如示例代码 13-1 所示。

示例代码 13-1

```
//表示一本书的类
public class Book
{
    public Book(string id, string name) //创建一本书
    {
        this.ID = id;
        this.Name = name;
    }
    public string ID                    //获取书的编号
    {
        get;
    }
    public string Name                  //获取或设置书名
    {
        get;
        set;
    }
    public string Author                //获取或设置作者
    {
        get;
        set;
    }
    public double Price                 //获取或设置价格
    {
        get;
        set;
    }
    public DateTime Date                //获取或设置日期
    {
        get;
        set;
    }
}
```

注意：并非一定要一个独立的类来表示表格中行的数据，每一行数据可以来自任何合法的地方，比如多个类、文件、数据库等。

在 DataGridView 控件中，可以通过 Rows 属性获取当前表格中所有的行信息，可以通过 CurrentRow 获取当前选中的行。通过 DataGridViewRow 类的 Cells 属性可以获取当前行所有的单元格信息。

在本例中，需要添加 3 个按钮，分别用来触发添加、删除和更改书籍信息的功能，它们的具体实现原理如下，具体实现如示例代码 13-2 所示。

- 添加书籍：创建一个新的 Book 对象 book，然后通过 Rows.Add()方法添加新行 newRow，并将 book 的数据显示到 newRow 中。另外，将 book 保存到 newRow 的 Tag 属性中。
- 删除书籍：通过 CurrentRow()方法获取当前选中行，然后通过 Rows.Remove()方法删除该行。
- 修改书籍：通过 CurrentRow()方法获取当前选中行，然后通过它的 Tag 属性获取对应的 Book 对象 book，之后修改 book 的值，最后将 book 更新到当前选中行中。

示例代码 13-2

```
public partial class FrmMain : Form
{
    private void FrmMain_Load(object sender, EventArgs e)
    {
        //不允许用户通过表格添加或删除记录
        this.dgvBooks.AllowUserToAddRows = false;
        this.dgvBooks.AllowUserToDeleteRows = false;
        //整行显示，而且只能单行选中
        this.dgvBooks.SelectionMode = DataGridViewSelectionMode.
        FullRowSelect;
        this.dgvBooks.MultiSelect = false;
    }
    private Random m_Random = new Random(100);
    private Book CreateBook( )
    {
        //生成一个随机数，用来创建书的信息
        int seed = m_Random.Next(1, 100);
        string bookName = string.Format("BookName-{0}", seed);
        //创建一本新书
        Book book = new Book(seed.ToString("D3"), bookName);
        //设置书的其他信息
        book.Date = DateTime.Today.AddDays(seed);
        book.Price = seed;
        book.Author = string.Format("Author-{0}", seed);
        return book;
    }
    private void ModifyBook(Book book)
    {
        //生成一个随机数，用来创建书的新信息
        int seed = m_Random.Next(1, 100);
        //设置书的新信息
        book.Date = DateTime.Today.AddDays(seed);
        book.Price = seed;
        book.Author = string.Format("Author-{0}", seed);
    }
```

```
    private void ShowBook(Book book, DataGridViewRow row)
    {
        //依次显示行中每一个单元格的数据，通过列名访问更可靠
        row.Cells["colBookID"].Value = book.ID;
        row.Cells["colBookName"].Value = book.Name;
        row.Cells["colPrice"].Value = book.Price;
        row.Cells["colAuthor"].Value = book.Author;
        row.Cells["colDate"].Value = book.Date.ToString("yyyy-MM-dd");
        //将当前行表示的书保存在 Tag 属性中
        row.Tag = book;
    }
    private void btnAddBook Click(object sender, EventArgs e)
    {
        //创建一本书
        Book book = CreateBook( );
        //创建新行，返回新建行的索引
        int newRowIndex = this.dgvBooks.Rows.Add( );
        //通过新建行的索引，获取新建的行
        DataGridViewRow newRow = this.dgvBooks.Rows[newRowIndex];
        //显示数据到新行
        ShowBook(book, newRow);
    }
    private void btnDelete Click(object sender, EventArgs e)
    {
        //获取当前选中行，如果不为 null，则删除选中行
        if (this.dgvBooks.CurrentRow != null)
            this.dgvBooks.Rows.Remove(this.dgvBooks.CurrentRow);
    }
    private void btnModify_Click(object sender, EventArgs e)
    {
        if (this.dgvBooks.CurrentRow == null)
            return;
        //获取当前行表示的书
        Book book = this.dgvBooks.CurrentRow.Tag as Book;
        if (book != null)
        {
            //修改书籍信息
            ModifyBook(book);
            //重新显示书籍信息到界面
            ShowBook(book, this.dgvBooks.CurrentRow);
        }
    }
}
```

生成实例 UnboundDataGridView，并添加多行数据之后，得到如图 13-14 所示的运行效果。值得注意的是，所有和外观相关的配置信息都是在属性管理器中实现，并没有编写任何代码。

13.3.5　绑定 DataGridView 的数据

通过前面几节的学习知道，在非绑定模式下使用 DataGridView 控件时，创建和编辑列信息会很重要，但是比较烦琐。所以当要显示和编辑大量数据时，通常通过数据绑定来加快这一动作。通过数据绑定，Visual Studio 2010 可以根据数据源自动为 DataGridView 控件创建必需的列信息，在运行时，DataGridView 控件可以自动从数据源获取并显示数据。



图 13-14　UnboundDataGridView 实例运行效果

在 DataGridView 控件中，属性 DataSource 和 DataMember 用来确定当前绑定的数据源。其中，DataSource 属性表示数据绑定的数据源，类似于一个小型的数据库，也可以是一个 BindingSource 数据源。而 DataMember 属性则表示真正要绑定和显示的 DataSource（数据源）中的成员，类似于数据库中的表。

在数据绑定模式下使用 DataGridView 控件必须要设置 DataSource 属性，而且通常为一个 BindingSource 控件。实例 BoundDataGridView 演示在数据绑定模式下使用 DataGridView 控件的常见步骤，大致如下。

（1）创建一个 Windows 窗体应用程序，命名为 BoundDataGridView，并重命名默认窗体 Form1 为 FrmMain，并设置它的标题。

（2）从工具箱拖一个 DataGridView 控件到窗体上，并重命名为 dgvUsers，根据需要设置它的外观。

（3）从工具箱拖一个 BindingSource 控件到窗体上，并重命名为 bsUserLogs，设置 bsUserLogs 的数据源为数据库 UserLog 的表 Users 和 Logs，设置它的 DataMember 属性为表 Users，如图 13-15 所示。

（4）在属性管理器中设置控件 DataGridView 的 DataSource 属性，为第 3 步创建的 bsUserLogs。

（5）此时，从设计器中查看 DataGridView 控件 dgvUsers 可以看到，Visual Studio 2010 已经根据数据源自动为它创建了列信息，如图 13-16 所示。

（6）从图 13-16 中可以看出，Visual Studio 2010 自动将数据表中所有的列一对一映射到 DataGridView 控件的列，并且将列名作为列标题。所以，通常需要重新编辑列的标题，使得标题可读性更好。

（7）到此，生成并运行实例 BoundDataGridView，可以看到 DataGridView 控件已经可以自动加载并显示数据库中的记录，如图 13-17 所示。

技巧： 在没有创建 BindingSource 的情况下，可以在属性管理器中直接设置 DataGridView 控件的 DataSource 属性，在列表中选择新建数据源功能，Visual Studio 2010 会自动创建 BindingSource 控件，并且 DataSource 属性的值为新建的 BindingSource 控件。



图 13-15　选择数据源

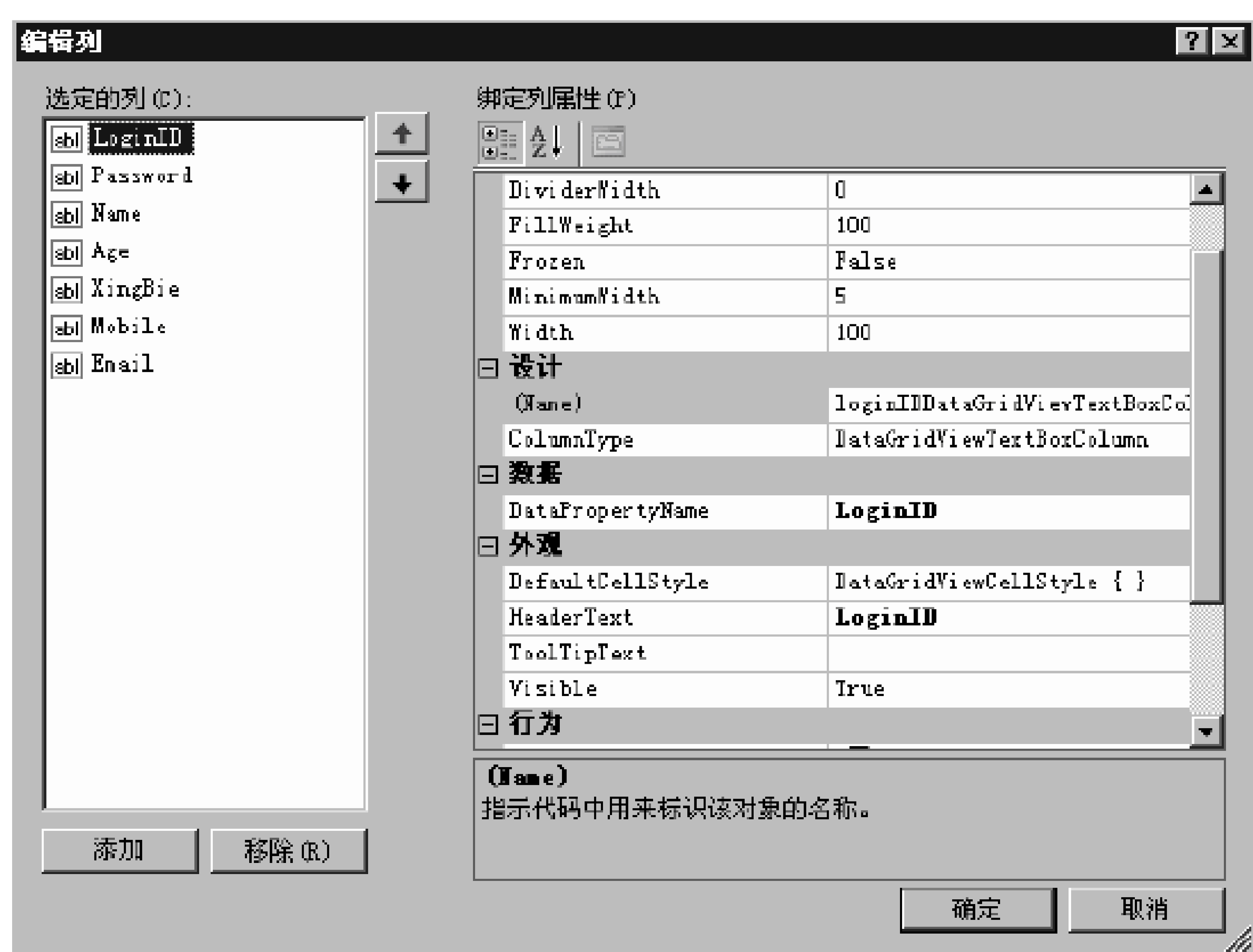


图 13-16　自动创建的列信息



图 13-17　BoundDataGridView 运行效果

13.3.6　用 DataGridView 修改数据

从 13.3.5 节的实例可以看出，通过数据绑定的形式使用 DataGridView 非常容易，在 Visual Studio 2010 的帮助下，甚至不需要任何编码，只需要拖放一些控件即可。但是，DataGridView 控件不仅可以显示数据，它同样可以编辑数据。

DataGridView 控件包括 3 个编辑功能：添加（Add）、删除（Delete）和修改（Modify），分别用 3 个属性来表示是否允许用户执行这些操作。

- ❑ AllowUserToAddRows：bool 类型读写属性，表示 DataGridView 控件是否允许用户添加一行新的数据，true 表示允许，false 表示不允许。如果某个列有指定的默认值，则新添加的行中该列自动使用默认值，否则用空。
- ❑ AllowUserToDeleteRows：bool 类型读写属性，表示 DataGridView 控件是否允许用户删除当前选中行，true 表示允许，false 表示不允许。按下 Delete 键即可删除当前选中行。
- ❑ ReadOnly：bool 类型读写属性，表示 DataGridView 控件是否允许用户编辑当前选中单元格的数据，true 表示不允许，false 表示允许。通过双击单元格可以进入编辑状态。

注意：DataGridView 控件的列定义中也有一个 ReadOnly 属性，也就是每个列可以单独设置是否只读，但是以大控件为主。即，如果 DataGridView 设置为只读，则所有列都为只读。

在 DataGridView 控件中，本质上是使用非连接模式下的 ADO.NET 进行数据库操作，所以对数据进行编辑只是编辑内存中的 DataSet，而不是直接将数据更新到数据库服务器。因此，需要开发人员自己编写代码将数据更改提交到服务器，这和使用 ADO.NET 更新数据库一样，通过 DataSet 类的 RejectChanges()方法取消已经应用的更改，通过 DataAdapter 类的 Update()方法提交更改到数据库服务器。

为了演示如何将 DataGridView 控件中的数据更改提交到数据库，这里为 13.3.5 节的实例 BoundDataGridView 增加两个功能：取消更改和提交更改。提交更改是将 DataGridView 上的更改提交到数据库，相反地，取消更改则是放弃 DataGridView 控件中的更改。大致需要以下步骤：

（1）在 Visual Studio 2010 中打开实例 BoundDataGridView。

（2）在窗体 Load 事件处理函数中设置 DataGridView 控件 dgvUsers 为允许添加、删除和非只读。

（3）为窗体 FrmMain 添加两个按钮，分别命名为 btnRollback 和 btnSubmit，标题分别为 “取消更改”和“提交更改”。

（4）在 btnRollbakc 的 Click 事件处理函数中添加回滚更改的代码。

（5）在 btnSubmit 的 Click 事件处理函数中添加提交更改的代码。

最终，BoundDataGridView 控件的实现如示例代码 13-3 所示，FrmMain_Load()方法通过设置 DataGridView 控件的 AllowUserToAddRows、AllowUserToDeleteRows 和 ReadOnly 属性来允许用户修改控件中的数据。btnRollback_Click()方法则通过 userLogDataSet 的 RejectChanges()方法拒绝修改，即取消更改。btnSubmit_Click()方法则通过 usersTableAdapter 的 Update()方法提交数据集合 userLogDataSet 中的更改到数据库。

示例代码 13-3

```
public partial class FrmMain : Form
{
   public FrmMain( )
   {
      InitializeComponent( );
   }
   private void FrmMain Load(object sender, EventArgs e)
   {
      //加载数据，本行为自动添加
      this.usersTableAdapter.Fill(this.userLogDataSet.Users);
      //允许添加，允许删除，允许编辑
      this.dgvUsers.AllowUserToAddRows = true;
      this.dgvUsers.AllowUserToDeleteRows = true;
      this.dgvUsers.ReadOnly = false;
   }
   private void btnRollback Click(object sender, EventArgs e)
   {
      //回滚所有的更改
      this.userLogDataSet.RejectChanges( );
   }
   private void btnSubmit Click(object sender, EventArgs e)
   {
```

```
        //提交数据更改到数据库
        this.usersTableAdapter.Update(this.userLogDataSet);
    }
}
```

生成并运行实例 BoundDataGridView，可以得到如图 13-18 所示的运行效果，修改之后单击“取消更改”按钮，可以发现更改消失，单击“提交更改”按钮，会发现修改被保存到数据库。



图 13-18　BoundDataGridView 运行效果

提示： 前面代码中的 userLogDataSet 和 usersTableAdapter 都是在添加数据源时自动生成的，它们都是在无连接模式下 ADO.NET 的必备成员，也是由它们真正去执行数据库操作。

13.4　小　　结

数据绑定是一种将数据源和控件联系到一起的技术，通过该技术，控件可以自动检查到数据源的更改，同时更新界面上显示的数据，得到数据和界面的同步。同时还可以通过数据绑定技术对数据源中的数据进行修改。通过 Visual Studio 2010 开发环境提供的数据源创建和管理功能，可以大大提高开发效率。

本章介绍了.NET 数据绑定的基本知识，通过本章的学习读者应该掌握以下知识点：

- 为什么要使用数据绑定？它有什么好处？
- Windows 窗体数据绑定的基本原理。
- BindingSource 在数据绑定中的角色，以及它的基本成员。
- 如何通过 BindingNavigator 进行数据导航？
- 如何将数据绑定到简单的控件，如 TextBox、Label 等？
- DataGridView 控件有哪些功能？有哪些主要成员？
- 如何设置 DataGridView 控件的外观？
- 如何在非数据绑定模式下操作和访问 DataGridView 控件上的数据？
- 如何为 DataGridView 控件添加数据绑定？
- 如何在数据绑定模式下回滚或提交 DataGridView 控件的数据更改？

第 5 篇　LINQ 查询开发

第 14 章　LINQ 查询基础

随着软件日益复杂化，软件开发所需要处理的数据无论是规模还是复杂度都急剧增大，对内存中大量数据的查询变得日益重要。于是，软件领域需要一种技术，它能够将高级编程语言和数据查询无缝集成到一起，使得内存中的数据查询变得更加高效，开发更加快速。语言集成查询 LINQ（Language Integrate Query）就是这样一种技术，它是.NET 4.0 推出的一个非常重要的特性。本书接下来几章将全面介绍 LINQ。

14.1　了解 LINQ 基础概念

LINQ 作为.NET Framework 4.0 的主要特性之一，为开发人员提供统一的查询内存数据的开发模式，将查询语言与.NET 高级编程语言（如 C#和 VB.NET）集成，很大程度上简化数据查询的编码和调试等工作，大大提高了开发效率。

14.1.1　什么是 LINQ

随着电脑和应用软件技术的不断普及，软件的应用环境越来越多样化，软件需要处理的数据量也日渐庞大，数据之间的关系越来越复杂。开发人员也逐渐需要对内存中的数据进行查询和分析，而且逻辑越来越复杂，处理的数据量也不断加大，甚至是一些模拟的内存数据库。所以，开发人员不仅要对外部数据（如数据库）进行查询和分析，也需要对内存数据进行类似的甚至更加复杂的处理。

传统的数据查询语言都定位于服务器端数据库查询，并不能对内存中的数据进行处理，而且其语法简单很难编写复杂的逻辑。SQL 查询语言作为标准的数据库查询语言，只是用简单的字符串文本来编写查询语句，没有编译时的类型检查，安全性、方便性都不好。此外，开发人员还需要学习不同的数据库提供商提供的 SQL 语言扩展，比如查询 SQL 数据库的 T-SQL，查询 XML 数据的 DOM 结构等。

为了让开发人员能够更加方便地对内存数据进行查询和分析，.NET 4.0 推出一项具有突破性的新特性——语言集成查询（LINQ）。LINQ 并不是独立的查询语言，它是一种更加简洁易懂的查询表达式编写语法，开发人员可以减少大量的 foreach 等代码。另外，.NET 4.0 还提供了多种与 LINQ 相结合的组件，这些组件在对象和数据之间建立一种对应关系，可以使用访问内存对象的方式查询数据集合，通常所说的 LINQ 是指这一整套组件的集合。

LINQ 使得查询表达式成为 C#中的一种语言元素，开发人员可以在 C#代码中嵌套类似于 SQL 语句的查询表达式，从而实现数据查询功能。但是，LINQ 并不是简单的在 C#中嵌套查询表达式，而是将查询表达式作为 C#的一个对象，编译时将对查询表达式进行类型检

查，增强了类型安全性，生成二进制代码，大大提高了执行效率。

LINQ 查询表达式本身也是一个对象，所以可以像使用普通的类对象那样使用它的字段和属性，也可以调用它的方法，同样可以响应它的事件。当然，也还可以通过 Visual Studio 2010 提供的智能感知功能来提高开发速度。对于复杂的查询逻辑，可以将查询逻辑实现为函数，然后在查询是调用该方法，从而简化查询表达式的开发。

LINQ 查询的数据源是内存中的对象集合，这些集合实现了统一的接口，使得 LINQ 查询能够按照统一的方式查询所有类型的数据。在.NET 4.0 中，LINQ 相关类库都在 System.Linq 命名空间下，该命名空间提供支持使用 LINQ 进行查询的类和接口，其中最主要的是两个类和两个接口。

- ❑ IEnumerable<T>：泛型接口，它定义可以被查询的数据集合，一个查询通常是逐个对集合中的元素进行筛选，返回一个新的 IEnumerable<T>对象，这个新的对象就是查询结果。
- ❑ IQueryable<T>：泛型接口，它继承自 IEnumerable<T>接口，定义一个可以查询的表达式目录树。
- ❑ Enumerable：静态类，为 IEnumerbale<T>提供大量扩展方法，从而支持 LINQ 标准查询运算符，包括：过滤、导航、排序、查询、联接、求和、求最大值、求最小值等。
- ❑ Queryable 类：静态类，为 IQueryable<T>提供大量扩展方法，从而支持 LINQ 标准查询运算符，包括过滤、导航、排序、查询、联接、求和、求最大值、求最小值等。

注意：深入学习 LINQ 之前，读者应该具备 LINQ 所用到的 C#语言特性，包括接口、泛型、扩展方法、可变类型、匿名类型等。

14.1.2　LINQ 有哪些相关组件

LINQ 作为.NET 4.0 的特性，主要实现对内存数据的查询和分析，它的另外一个目标是提供统一的编程方式来分析其他数据。.NET 4.0 提供多个与 LINQ 相关的组件，这些组件将内存对象和外部数据联系到一起，从而使得 LINQ 能够操作外部数据。如图 14-1 所示，主要包括以下组件。

- ❑ LINQ to Object：LINQ 的核心组件，将查询表达式与 C#编程语言集成，实现对内存对象表示的数据源进行查询和分析。数据源为实现了接口 IEnumerable<T>或 IQueryable<T>的内存数据集合，本章将详细介绍这方面的内容。
- ❑ LINQ to ADO.NET：将 LINQ 与 ADO.NET 集成，借助 ADO.NET 实现对多种数据库数据的查询和分析。包括 LINQ to SQL 和 LINQ to DataSet 两部分，前者主要操作外部数据库，后者操作内存 DataSet。
- ❑ LINQ to XML：一种全新的 XML 数据操作模式，与 DOM 有相似之处，但又完全不同。数据源为 XML 文档，通过 XElement、XAttribute 等类将 XML 文档数据加载到内存中，通过 LINQ 进行数据查询。

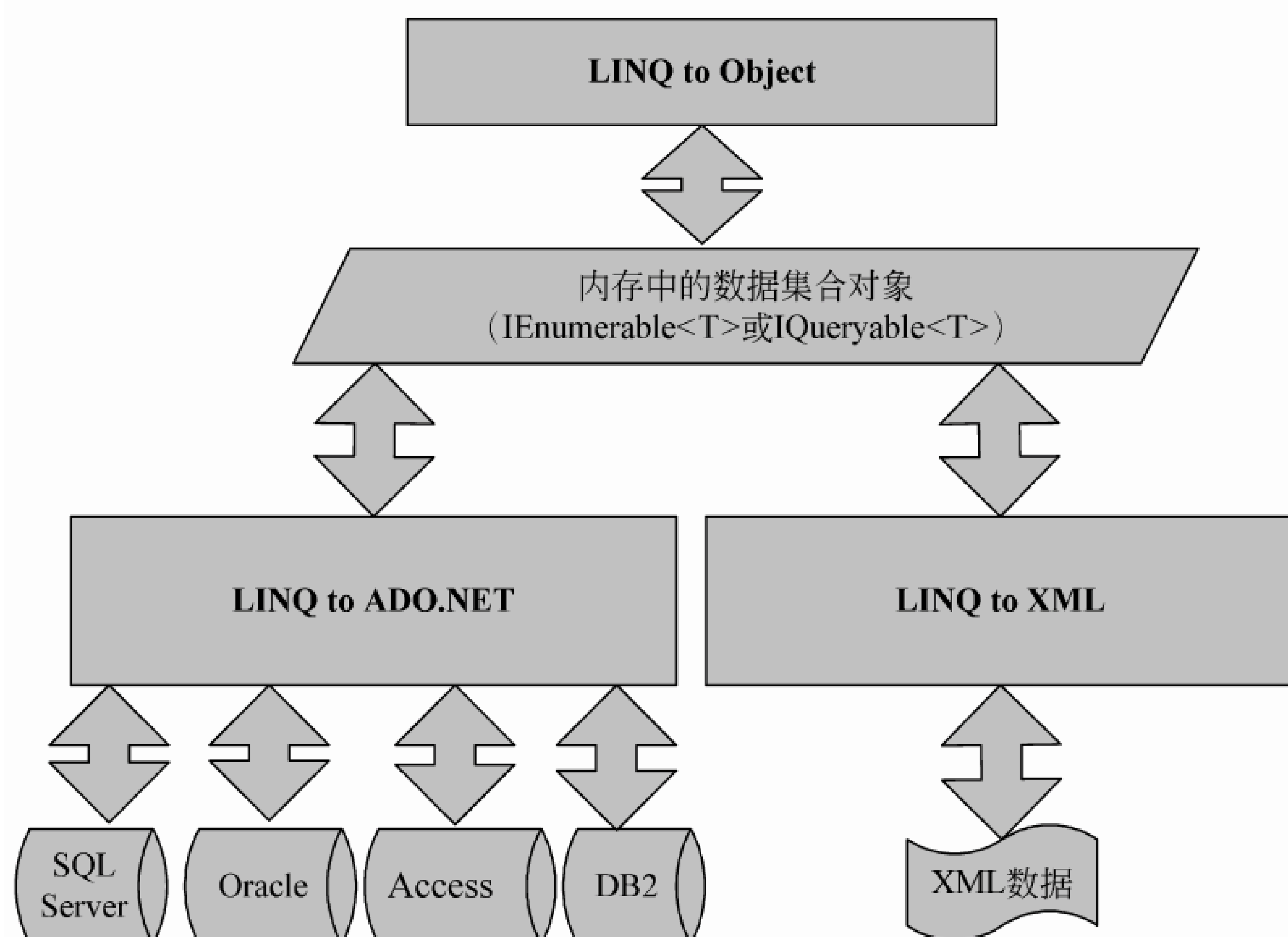


图 14-1　LINQ 相关组件

通过这 3 个组件，LINQ 可以查询数据库、XML 数据和内存数据集合。除此之外，.NET 4.0 还为用户扩展 LINQ 提供了支持，用户可以根据需要实现第三方的 LINQ 支持程序，然后通过 LINQ 获取自定义的数据源。所以，LINQ 本身并不关心数据来自哪里，它可以查询任何类型的数据，只要这些数据都被成功加载到内存即可。

14.1.3　LINQ 与 C#集成开发

LINQ 作为.NET 编程语言的一个扩展，可以与 C#、VB.NET 等.NET 开发语言集成使用，但是语法上有点类似于 SQL 语言。LINQ 直接嵌入到 C#语言中使用，就好像它是 C#语言的一个部分。LINQ 在编写数据源查询表达式时显得更加简单易懂。本节以一个简单例子展示 LINQ 与 C#的使用。

假如在某数值处理软件中，需要将整数数组中的所有元素增加 1，从而产生一个新的数组，并且对该数组按照从大到小的顺序排序。如示例代码 14-1 所示，函数 FuncInCSharp()就是通过 C#代码的形式来实现该功能，它主要通过了 3 个步骤，近 10 个语句实现该功能。函数 FuncInLINQ()同样是实现该功能，却只用了 1 个语句（3 行代码），由此可见，LINQ 使得这类代码更加简洁。

示例代码 14-1

```
class Program
{
    static void Main(string[] args)
    {
        int[] srcArray = new int[] {2, 6, 3, 8, 7, 1};
        FuncInCSharp(srcArray);
        FuncInLINQ(srcArray);
    }
    static void FuncInCSharp(int[] srcArray)
    {
```

```
        //1.创建新的数组
        int[] resArray = new int[srcArray.Length];
        //2.遍历数组中所有元素，并增加 1
        for (int index = 0; index < srcArray.Length; index++)
        {
            int val = srcArray[index];
            resArray[index] = val + 1;
        }
        //3.对新的数组进行从小到大排序，然后再逆序
        Array.Sort(resArray);
        Array.Reverse(resArray);
        //4.打印数组
        System.Console.WriteLine("Output in FuncInCSharp()");
        foreach (int val in resArray)
        {
            System.Console.Write("{0}, ", val);
        }
        System.Console.WriteLine( );
    }
    static void FuncInLINQ(int[] srcArray)
    {
        //1.编写 LINQ 查询，指定元素+1 为新的元素，并逆序排序
        var resQuery = from val in srcArray
                       orderby val descending
                       select val + 1;
        //2.打印查询结果
        System.Console.WriteLine("Output in FuncInLINQ()");
        foreach (int val in resQuery)
        {
            System.Console.Write("{0}, ", val);
        }
        System.Console.WriteLine( );
    }
}
```

从 FuncInLINQ()方法中的 LINQ 查询中，可以看到 From、Select、OrderBy 等字句，它们都是 LINQ 的关键字，它们和 SQL 命令类似，而且从字面上就可以理解整个查询的基本功能，可见 LINQ 查询比 C#查询代码可读性更好。

另外，LINQ 查询只是一种特殊的语法，和 C#代码并没有本质的区别，所以可以像使用普通 C#类对象那样使用 LINQ 查询 resQuery。由于它是一个集合，所以可以通过 foreach 来遍历它的元素。另外，LINQ 查询结果通常使用匿名类型（用 var 关键字表示），这样做可以减少开发人员的工作，而且在一些情况下只能使用匿名类型。

14.1.4　可枚举泛型接口 IEnumerable<T>

在 LINQ 查询中，数据源是实现了泛型接口 IEnumerable<T>或 IQueryable<T>的类对象，这两个接口定义了作为数据源都必须支持的方法和集合，并通过两个静态类 Enumerable 和 Querable 为它们提供了大量用于支持 LINQ 编写的扩展方法。

IEnumerable<T>泛型接口支持在指定数据集合上进行迭代操作，它定义了一组扩展方法，用来对数据集合中的元素进行遍历、过滤、排序、搜索、定位等操作。Enumerable 类为 IEnumerable<T>接口提供了大量与 LINQ 查询相关的扩展方法。如表 14-1 列出了 IEnumerable<T>接口的主要成员及其功能。

表 14-1　IEnumerable<T>主要成员

成　　员	功　　能
Aggregate	对序列应用累加器函数，可以指定累加方法
Sum	计算序列中所有元素的和，返回值有int、long、float、double、decimal类型，并且可以指定元素到数值的映射方法
Average	计算序列中所有元素的平均值，返回值有int、long、float、double、decimal类型，并且可以指定元素到数值的映射方法
Max	计算序列中所有元素的最大值，返回值有int、long、float、double、decimal类型，并且可以指定元素到数值的映射方法
Min	计算序列中所有元素的最小值，返回值有int、long、float、double、decimal类型，并且可以指定元素到数值的映射方法
All	检查是否序列中所有元素都满足条件，可以指定条件判断方法，如果所有元素都满足条件返回TRUE，否则返回FALSE
Any	检查序列中是否有任何一个元素满足条件，可以指定条件判断方法，如果有一个以上（含一个）元素满足条件返回TRUE，否则返回FALSE
Contains	检查数据序列中是否包含特定的元素，可以指定相等比较方法
Count	返回序列中满足指定条件的元素的数量，可以指定条件判断方法
LongCount	返回序列中满足指定条件的元素的数量，可以指定条件判断方法
Cast	将IEnumerable中的元素转换为指定的数据类型
DefaultIfEmpty	返回序列中指定位置的元素，如果序列为空，则返回默认的元素值
ElementAt	返回序列中指定索引处的元素
ElementAtOrDefault	返回序列中指定索引处的元素，如果索引超出范围，则返回默认值
First	返回序列中满足指定条件的第一个元素，可以指定条件判断方法
FirstOrDefault	返回序列中满足指定条件的第一个元素，如果不存在则返回默认值，也可以指定条件判断方法
Last	返回序列中满足指定条件的最后一个元素，可以指定条件判断方法
LastOrDefault	返回序列中满足指定条件的最后一个元素，如果不存在则返回默认值，也可以指定条件判断方法
Single	返回序列中满足指定条件的唯一元素，如果不止一个元素满足条件会引发异常，可以指定条件判断方法
SingleOrDefault	返回序列中满足指定条件的唯一元素，如果不存在则返回默认值，如果不止一个元素满足条件会引发异常，可以指定条件判断方法
Reverse	反转序列中元素的顺序
Distinct	返回序列中不重复的元素的集合，可以指定相等比较方法
Concat	连接两个序列，直接首尾相连，返回结果可能存在重复数据
Except	获取两个元素集合的差集，可以指定相等比较方法
Intersect	获取两个元素集合的交集，可以指定相等比较方法
Union	获取两个元素集合的并集，可以指定相等比较方法
SequenceEqual	比较两个序列是否相等，可以指定相等比较方法
Where	根据指定条件对集合中元素进行筛选，返回满足条件的元素集合
Skip	跳过序列中指定数量的元素，然后返回剩余的元素
SkipWhile	跳过序列中满足指定条件的元素，然后返回剩余的元素，可以指定条件判断方法
Take	从序列的开头返回指定数量的连续元素
TakeWhile	返回从序列开始的满足指定条件的连续元素，可以指定条件判断方法
ToArray	从IEnumerable<T>创建一个数组
ToList	从IEnumerable<T>创建一个List<T>

从表 14-1 中可以看出，IEnumerable<T>提供的方法包括数值运算（Sum、Min、Max、Average）、元素数量（Count、LongCount）、取值（First、Last、ElementAt 等）、提取子集（Skip、SkipWhile、Take、TakeWhile）、集合操作（Reverse、Concat、Distinct、Except、Intersect、Union、SequenceEqual 等）。这些方法提供了 LINQ 查询所需要的所有操作，在后面几节中将介绍几个常用方法的具体使用。

值得注意的是 IEnuerable<T>继承至 IEnumerable 接口，所以它也包含 IEnumerable 接口的所有方法，所以还包括 Select()、SelectMany()、Repeat()等方法，在表 14-1 中没有给出。另外，IQueryable<T>接口从 IEnumerable<T>派生而来，通常也可以作为数据源使用，它的使用和 IEnumerable<T>类似，本书就不作进一步的介绍，可以参看本书关于 IQueryable<T>的介绍。

14.2　使用 LINQ 表达式查询

查询表达式是 LINQ 查询的核心元素，它和传统的查询语言编写方式类似，也是 LINQ 查询最直观、最简洁、最容易理解的编写方式。本节从 LINQ 查询表达式的各子句出发，逐步由浅入深逐步讲解 LINQ 查询表达式的语法和具体应用。

14.2.1　了解查询语法和查询表达式

在.NET 4.0 中，有两种语法用来编写 LINQ 语句：查询语法和方法语法。查询语法通过查询表达式的方式编写 LINQ 语句，如示例代码 14-1 中函数 FuncInLINQ()所示，有点类似于 SQL 语句。方法语法则是将 LINQ 查询作为对象，并通过 C#代码操作它们，从而达到查询的功能。本节介绍 LINQ 的查询语法，后面几节将介绍 LINQ 的方法语法。

查询表达式是 LINQ 查询语法的核心，也是构成 LINQ 查询语句的基本元素，同样也是最直接、最易懂、最常用的 LINQ 编写方式。查询表达式是由查询关键字和对应的操作数组成的表达式整体，其中查询关键字是常用的查询运算符，C#为这些运算符提供对应的关键字，从而更好地与 LINQ 集成。

每个查询关键字对应一个 LINQ 查询子句，各个子句具有明确的功能，将各子句组合到一起，就形成了一个完整的具有某种特定功能的 LINQ 查询。在 C# 3.0 中可以直接使用的查询子句及其功能如表 14-2 示，从中可以看出，它们和 SQL 有些类似，但却又不同。

表 14-2 常用的LINQ子句

子　句	功　能
from	指定要查找的数据源以及范围变量，多个from子句则表示从多个数据源中查找数据
select	指定查询要返回的目标数据，可以指定任何类型甚至是匿名类型
where	指定元素的筛选条件，多个where子句则表示了并列条件，必须全部都满足才能入选
orderby	指定元素的排序字段和排序方式，当有多个排序字段时，由字段顺序确定主次关系，可指定升序和降序两种排序方式
group	指定元素的分组字段
Join	指定多个数据源的关联方式

LINQ 查询的根本目的是从指定的数据源中查询满足符合特定条件的元素，并且根据需要对这些查询到的元素进行排序、连接等操作。所以，LINQ 查询包括几个主要元素。

- 数据源：数据源表示 LINQ 查询将从哪里查找数据，它通常是一个或多个数据集，每个数据集包含一系列的元素。数据集是一个类型为 IEnumerable<T>或 IQueryable<T>的对象，可以对它进行枚举，遍历每一个元素。此外，它的元素可以是任何数据类型，所以可以表示任何数据的集合。
- 目标数据：数据源中的元素并不全是查询所需要的结果，例如，对于一个学生信息集合中，查询 A 只是查询学生的姓名，查询 B 要查询学生的姓名和各科成绩，查询 C 则需要学生各科成绩的总分（需要另外计算），而不是原始数据中的各科成绩。目标数据用来指定查询具体想要的是什么数据，在 LINQ 中，它定义了查询结果数据集中元素的具体类型。
- 筛选条件：筛选条件定义了对数据源中元素的过滤条件，只有满足条件的元素才作为查询结果返回。筛选条件可以是简单的逻辑表达式表示，也可以用具有复杂逻辑的函数来表示。
- 附加操作：附加操作表示一些其他的对查询结果的辅助操作，比如，对查结果询进行排序、计算查询结果的最值和求和、对查询结果进行分组等。

数据源和目标数据是 LINQ 查询的必备元素，筛选条件和附加操作是可选元素。从表 14-2 中可以看出，from 子句在 LINQ 查询中是必不可少的，因为它指定了 LINQ 查询的数据源。而 select 子句和 group 子句指定了 LINQ 查询结果中的元素，所以它们必须有且只有一个存在。后面的内容中将进一步对各关键字进行介绍，包括它们的格式、使用方法、技巧等。

注意：在 SQL 查询中，关键字是大小写无关的。但是在 LINQ 查询代码中关键字是大小写有关的，所有关键字都是必须是小写，否则将出现编译错误，可以将它们作为 C#关键字来看。

14.2.2　用 from 子句指定数据源

数据源是 LINQ 查询中必不可少的元素，它表示 LINQ 查询要从哪里查找符合条件的元素。在 LINQ 查询中，数据源是实现了泛型接口 IEnumerable<T>或 IQueryable<T>的类对象，可以将 IEnumerable<T>简单理解成一个包含多个元素的列表（或数据库中的表），可以用 foreach 遍历它的所有元素，从而轻松完成查询操作。

由于 IEnumerable<T>是泛型接口，所以通过为数据源指定不同的元素类型，可以表示任何数据集合。在.NET 类库中，列表类、集合类、数组等都实现了接口 IEnumerable<T>，所以，可以直接将这些数据对象作为数据源在 LINQ 查询中使用。

LINQ 查询中，通常以 from 子句开始，from 子句指定查询将采用的数据源，同时定义一个本地变量，表示数据源中单个元素，整个 LINQ 查询都是对该元素进行操作。单个 from 子句的编写格式如下，其中，dataSource 表示数据源，localVar 是单个元素的本地变量。

```
from localVar in dataSource
```

一般情况下，不用为 from 子句的 localVar 元素指定数据类型，编译器会根据数据源类型为它分配合适的类型，通常元素类型为 IEnumerable<T>中的类型 T。例如，当数据源为 IEnumerable<int>时，编译器为 localVar 指定类型为 int，当数据源为 IEnumerable<string>时，编译器为 localVar 指定类型为 string。如示例代码 14-2 所示，由于 intAry 是 int[ ]类型，默认实现了接口 IEnumerable<int>，所以 intVal 的类型为 int。

示例代码 14-2

```
static void FuncIntSrc( )
{
    int[] intAry = { 78, 19, 100, 12, 23, 60 };
    var query1 =
        from intVal in ary
        select intVal;
}
```

一些特殊情况下，开发人员还需要为本地变量指定数据类型，比如示例代码 14-2 中，希望将 ary 中的元素作为 object 类型进行处理，而不是作为 int。这就需要在 from 子句中为 localVar 指定目标类型，格式如下，其中，localType 是本地变量 localVar 的类型。

```
from localType localVar in dataSource
```

如示例代码 14-3 所示，指定 objVal 为 object 类型，由于 ary 中的元素为 int 类型，属于 object 类型的子类型，所以可以直接转换为 object 类型。指定 dblVal 为 double 类型，由于 ary 中的元素为 int 类型，可以强制转换为 double 类型，所以可以直接转换为 double 类型。

示例代码 14-3

```
static void FuncObjectSrc( )
{
    int[] intAry = { 78, 19, 100, 12, 23, 60 };
    var query1 =
        from object objVal in ary
        select objVal;
}
static void FuncDoubleSrc( )
{
    int[] intAry = { 78, 19, 100, 12, 23, 60 };
    var query1 =
        from double dblVal in ary
        select dblVal;
}
```

为本地变量（localVar）指定数据类型时，需要注意编译器并不会检查本地变量的具体类型，所以当指定类型不正确时，编译时并不会报错。如示例代码 14-4 所示，本地参数 strVal 指定为 string 类型，但是 intAry 的元素实际是 int 类型，所以将它指定为 string 类型有错误，但编译器并不会报错。但是当使用该查询结果时，会在运行时进行类型检查，从而产生如图 14-2 所示的异常。

示例代码 14-4

```
static void FuncStringSrc( )
{
```

```
    int[] intAry = { 78, 19, 100, 12, 23, 60 };
    var query1 =
        from string strVal in ary
        select strVal;
}
```



图 14-2　类型错误对话框

建议：如果没有特别需要，作者建议使用不指定类型的本地变量，让编译器自动根据数据源判断具体的元素类型。

14.2.3　用 select 子句指定查询结果

在 LINQ 查询中，数据源及其元素都通过 from 子句来指定。查询结果本身则通过 select 或 group 子句来定义，其中 select 子句最常用，它指定的查询结果为一维的查询结果。select 子句的定义如下：

```
from localVar in dataSource
select expression
```

其中，expression 是一个表达式，用来计算要作为查询结果的元素。该表达式通常是对 from 子句中的 localVar 进行某种运算，最简单的一种就是直接使用 localVar，如下所示，这样产生的查询结果就包含与数据源（dataSource）一样的元素。

```
from localVar in dataSource
select localVar
```

理论上，expression 可以使任何合法的 C#语言表达式，可以对 localVar 进行计算，也可以创建新的对象作为查询结果的元素，如示例代码 14-5 所示。查询 valStrQuery：

示例代码 14-5

```
public class ValString
{
    //构造函数，传入 int 类型的值
    public ValString(int val)
    {
        Value = val;
    }
    public int Value
    {
        get;
        set;
```

```
    }
    //重写 ToString()方法，使得打印更方便
    public override string ToString()
    {
        return String.Format("Value={0:D5}", Value);
    }
}

class Program
{
    static void Main(string[] args)
    {
        SelectValString( );
    }
    static void SelectValString( )
    {
        //创建整数数组，作为数据源
        int[] intSrc = new int[] {2450, 3445, 19, 228, 33};
        //用数据源中所有的元素创建对应的 ValString 对象，并作为查询结果返回
        var valStrQuery = from intVal in intSrc
                          select new ValString(intVal);
        //依次打印查询结果中的所有元素
        foreach (ValString item in valStrQuery)
            System.Console.WriteLine(item);
    }
}
```

生成并运行示例代码 14-5，输出如下，从中可以看出，输出为 ValString.ToString()产生的字符串，可见查询结果 valStrQuery 中的元素类型为 ValString。

```
Value=02450
Value=03445
Value=00019
Value=00228
Value=00033
```

技巧：select 子句的 expression 可以是一个常量表达式，则查询结果中所有的元素都为这个常量值，如 select 5，则查询结果所有元素都为 5。查询结果中元素的个数和查询表达式有关。

14.2.4　在 select 子句中创建匿名类型

示例代码 14-5 中，首先为查询结果创建了一个类型 ValString，这样编码工作量明显增加，尤其是当该类型并没有多少实际用处，仅作为查询结果元素使用时，更加浪费。在这种情况下，通常需要在 select 子句中使用匿名类型。

在 select 子句中使用匿名类型非常简单，只需要在 expression 中创建需要的匿名类型即可，如示例代码 14-6 所示，查询 anonyQuery 中 select 子句通过 new 关键字直接创建一个匿名类型，该类型包括两个属性 IntValue 和 DoubleValue，其中 DoubleValue=IntValue*1.5。

示例代码 14-6

```
class Program
{
```

```
    static void Main(string[] args)
    {
        SelectAnonymous( );
    }
    static void SelectAnonymous( )
    {
        //创建整数数组，作为数据源
        int[] intSrc = new int[] { 2450, 3445, 19, 228, 33 };
        //用数据源中所有的元素创建一个匿名类型
        var anonyQuery = from intVal in intSrc
                         select new {IntValue = intVal, DoubleValue=
                         intVal*1.5};
        //依次打印查询结果中的所有元素
        foreach (var item in anonyQuery)
            System.Console.WriteLine(item);
    }
}
```

示例代码 14-6 的输出如下，从中可以看出，匿名类型在 LINQ 查询临时使用时非常实用。

```
{ IntValue = 2450, DoubleValue = 3675 }
{ IntValue = 3445, DoubleValue = 5167.5 }
{ IntValue = 19, DoubleValue = 28.5 }
{ IntValue = 228, DoubleValue = 342 }
{ IntValue = 33, DoubleValue = 49.5 }
```

值得注意的是，由于 select 查询产生的元素是匿名类型，所以在定义查询结果和使用查询结果时都只能使用匿名类型，因为查询结果是依赖于查询结果中元素的类型而定的。

14.2.5　用 where 子句指定过滤条件

前面的 LINQ 查询中，都是直接对数据源中所有元素进行查询，显然很多开发应用场景都不会如此简单。对数据源中数据的过滤是一个十分常见的应用，也是作为查询最基本的功能之一。在 LINQ 中，通过 where 子句指定对元素进行过滤的条件，where 子句的位置在 from 之后 select 之前，如下所示。

```
from localVar in dataSource
where condition
select expression
```

其中，condition 表示过滤条件，是任何类型为 bool 的 C#表达式，如果值为 true，则该元素符合条件，将用来产生查询结果，否则过滤该元素。

condition 通常是对 localVar 进行逻辑运算，从而判断它是否满足条件，该表达式可以非常简单，甚至是一个 true 或 false 的常量。condition 也可以比较复杂，比如，多个逻辑表达式进行表 14-3 所示的逻辑运算，甚至可以执行某个函数得到该条件。

表 14-3　where子句中的逻辑操作

exp1	exp2	exp1 && exp2	exp1 || exp2	!exp1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

如示例代码 14-7 所示，SimpleWhere()方法中，查询 query1 的 where 子句就是一个简单的条件判断，而查询 query2 的 where 子句就是两个简单逻辑判断进行&&（与）操作。查询 query3 的 where 子句，则通过调用函数 IsPrimeNumber()判断元素 intVal 是否满足条件。

示例代码 14-7

```
class Program
{
    static void Main(string[] args)
    {
        SimpleWhere( );
    }

    static void SimpleWhere( )
    {
        //创建整数数组作为数据源
        int[] intSrc = new int[] {10, 13, 22, 32, 15, 27, 37, 48, 89};
        //定义查询，只需要大于 15 的元素
        var query1 = from intVal in intSrc
                     where intVal > 15
                     select intVal;
        //打印查询 query1 结果
        System.Console.Write("Query1: ");
        foreach (var item in query1)
            System.Console.Write("{0}, ", item);

        //定义查询，只需要大于 15 并小于 40 的元素
        var query2 = from intVal in intSrc
                     where (intVal > 15) && (intVal < 40)
                     select intVal;
        //打印查询 query2 的结果
        System.Console.WriteLine( );
        System.Console.Write("Query2: ");
        foreach (var item in query2)
            System.Console.Write("{0}, ", item);

        //定义查询，只需要数据源中所有的质数
        var query3 = from intVal in intSrc
                     where IsPrimeNumber(intVal)
                     select intVal;
        //打印查询 query3 的结果
        System.Console.WriteLine( );
        System.Console.Write("Query3: ");
        foreach (var item in query3)
            System.Console.Write("{0}, ", item);
    }
    static bool IsPrimeNumber(int val)
    {
        if (val <= 1)
            return false;
        for (int i = 2; i < val; i++)
        {
            if (val % i == 0)
                return false;
        }
        return true;
    }
}
```

示例代码 14-7 的输出如下所示。

```
Query1: 22, 32, 27, 37, 48, 89,
Query2: 22, 32, 27, 37,
Query3: 13, 37, 89,
```

技巧： 当 where 子句的条件比较复杂时，应尽可能地通过函数的形式实现条件，而在 where 子句中调用该函数，这样可以让代码更加简洁，且可读性更好。

14.2.6　用并列 where 子句指定多个条件

除了用函数来实现判断条件之外，还有一种比较简单的方式让 where 子句的可读性更好，那就是采用多个并列的 where 子句。在 LINQ 中，同一个查询中使用多个并列的 where 子句，这些 where 子句的条件按照与（&&）进行运算。

如示例代码 14-8 所示，方法 MultiWhere()中的查询 query，是对示例代码 14-7 中查询 query2 等价的两种写法，这里的两个并列的 where 子句效果上等价于将两个条件进行与（&&）运算。

示例代码 14-8

```
static void MultiWhere( )
{
    //创建整数数组作为数据源
    int[] intSrc = new int[] { 10, 13, 22, 32, 15, 27, 37, 48, 89 };

    //定义查询，只需要大于 15 并小于 40 的元素
    var query = from intVal in intSrc
                where intVal > 15
                where intVal < 40
                select intVal;
    //打印查询 query 的结果
    System.Console.Write("Query: ");
    foreach (var item in query)
        System.Console.Write("{0}, ", item);
}
```

当 where 子句本身不是很复杂，而且可以分解成多个简单的并列判断条件时，可以使用多个并列的 where 子句来实现，这样的写法比使用函数编写更加直观，但是 where 子句个数不易太多。

14.2.7　用 orderby 子句实现排序

对查询结果中的元素进行排序是常见的操作之一，在 LINQ 中，通过 orderby 子句对查询结果进行排序。orderby 子句可以对查询结果进行升序或降序排序，它的格式如下：

```
orderby expression [sortType]
```

其中，expression 是要进行排序的表达式，它可以是 from 子句中的 localVar，也可以是对 localVar 进行运算的表达式。sortType 是可选参数，表示排序类型，包括升序（ascending）

和降序（desending）两个可选值，默认情况下为升序。

如示例代码 14-9 中，查询 query1 是查询数据源 intSrc 中的所有元素，并按照从小到大排序。查询 query2 查询数据源 intSrc 中的所有元素，并按照从大到小排序。

示例代码 14-9

```
static void Main(string[] args)
{
    SimpleOrderBy( );
}
static void SimpleOrderBy( )
{
    //创建整数数据源
    int[] intSrc = new int[] {1, 3, 2, 9, 5, 7, 4, 8, 6};
    //创建查询，对数据进行从小到大排序
    var query1 = from intVal in intSrc
                 orderby intVal ascending
                 select intVal;
    //打印查询 query1 的结果
    System.Console.Write("Query1: ");
    foreach (var item in query1)
        System.Console.Write("{0}, ", item);
    //创建查询，对数据进行从大到小排序
    var query2 = from intVal in intSrc
                 orderby intVal descending
                 select intVal;
    //打印查询 query1 的结果
    System.Console.WriteLine( );
    System.Console.Write("Query2: ");
    foreach (var item in query2)
        System.Console.Write("{0}, ", item);
}
```

示例代码 14-9 的输出如下，query1 的查询结果按照升序排序。query2 的查询结果则按照降序排序，指定为 desending。

```
Query1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
Query2: 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

在 LINQ 中，orderby 子句可以同时指定多个排序元素，还可以为每个排序元素指定独立的排序方式，orderby 语句后的第一个的排序元素为主要排序，第二个为次要排序，依次类推。如示例代码 14-10 所示，首先按照 intVal%3 的值从大到小排序，如果 intVal%3 的值相等，则按照 intVal 从小到大排序。

示例代码 14-10

```
static void MultiOrderBy( )
{
    //创建整数数据源
    int[] intSrc = new int[] { 1, 3, 2, 9, 5, 7, 4, 8, 6 };
    //创建查询，对数据按照模 3 的值进行排序
    var query = from intVal in intSrc
                orderby intVal % 3 descending, intVal ascending
                select intVal;
    //打印查询 query 的结果
    System.Console.Write("Query: ");
```

```
    foreach (var item in query)
        System.Console.Write("{0}, ", item);
}
```

示例代码 14-10 的输出如下，由于数字 2、5、8 模 3 都为 2，而且为最大，所以被排在前面，它们 3 个数字按照从小到大的顺序进行排序。从中可以看出工分成了{2，5，8}，{1，4，7}，{3，6，9}三组数字。

```
Query: 2, 5, 8, 1, 4, 7, 3, 6, 9,
```

注意：orderby 子句和 where 子句不一样，当在一个 LINQ 查询中出现多个 orderby 子句时，只有最后一个 orderby 子句有效，前面的 orderby 子句都无效。

14.2.8　用 group 子句实现分组

在示例代码 14-10 中，如果可以对元素进行按照模 3 的值进行分组，查询结果会更加容易理解。在 LINQ 中可以通过 group 子句对查询结果进行分组。group 子句的常用格式如下：

```
group expression by key
```

其中，expression 是一个表达式，用于计算产生查询结果中的元素。而 key 同样是一个表达式，用于计算进行分组的条件。group 子句查询返回类型为 IGrouping<TKey,TElement>的查询结果，其中 TKey 的类型为表达式 key 的数据类型，TElement 的类型是表达式 element 的数据类型。

简单地说，可以将 IGrouping<TKey,TElement>看成一个哈希表，哈希表的每一个元素都是一个列表，它包括一个类型为 TKey 的属性 Key，表示查询中进行分组的关键字。通过关键字 Key 可以找到属于这个组的所有元素。所以，通常使用两层 foreach 遍历 IGrouping 中的所有元素，外层 foreach 按照 Key 遍历所有的分组，内层 foreach 遍历该分组的元素列表，从而得到结果中的所有元素。

在示例代码 14-11 中，查询 query 通过 group 子句对数据源中的元素按照模 3（%3）的值进行分组。在 group 子句“group intVal by intVal%3”中，intVal%3 是用来分组的表达式，它的类型为 int，由于 intVal 为分组中的元素，类型也为 int 类型，所以查询结果的类型为 IGrouping<int, int>。另外，在打印查询结果时，首先通过一个 foreach 语句从查询中找到所有的分组（局部变量 grp），然后对每个分组，使用 foreach 语句查找分组中的元素（局部变量 val）。

示例代码 14-11

```
class Program
{
    static void Main(string[] args)
    {
        SimpleGroupBy( );
    }
    static void SimpleGroupBy( )
    {
```

```
        //创建整数数据源
        int[] intSrc = new int[] { 1, 3, 2, 9, 5, 7, 4, 8, 6 };
        //创建查询，对数据按照模 3 的值进行分组
        var query = from intVal in intSrc
                    orderby intVal
                    group intVal by intVal % 3;
        //打印查询 query 的结果
        System.Console.WriteLine("Query: ");
        //外层循环遍历所有的分组
        foreach (var grp in query)
        {
            System.Console.Write("%3={0}的元素有: ", grp.Key);
            //内层循环遍历当前分组中的所有元素
            foreach (var val in grp)
                System.Console.Write("{0}, ", val);
            System.Console.WriteLine( );
        }
    }
}
```

示例代码 14-11 的输出如下，从中可以看出，所有的元素被按照模 3 的值进行了分组，而且同一组中元素出现的顺序与它们在数据源中的顺序保持一致。

```
Query:
%3=1 的元素有: 1, 4, 7,
%3=0 的元素有: 3, 6, 9,
%3=2 的元素有: 2, 5, 8,
```

在一个 LINQ 查询中，对于这种简单格式的 group 子句只能出现一个，而且出现在 LINQ 查询的末尾，并且，不能和 select 子句同时存在。示例代码 14-11 中就是这种用法，其有诸多局限，而且使用太简单，14.2.9 节将介绍 group…into 子句的使用，用它可以将分组作为数据源再次查询。

14.2.9　用 into 子句缓存查询结果

从示例代码 14-11 的输出可以看出，虽然元素被分组，并且每个分组中的元素都被排序，但是分组本身并没有进行排序，看起来有点杂乱无章。而且在很多实际开发中，都需要对查询结果对分组进行重新过滤、排序等操作。这就需要使用 group…into 子句，本节介绍该子句的详细使用。

在 LINQ 中，into 子句表示将本次查询产生的结果缓存到一个临时变量，从而可以把它作为数据源再次进行查询。所以 into 子句本身并不能独立使用，它要与 select 或 group 一起使用，格式如下：

```
select expression into tmpSource
group expression by key into tmpSource
```

其中，tmpSource 是一个变量，查询结果就保存在该变量中。在当前查询的后面子句中，可以继续对查询 tmpSource 进行任何 LINQ 查询操作，比如排序、过滤、select、group 等。

如示例代码 14-12 所示，在示例代码 14-11 的基础上，通过 into 子句将 group 子句产

生的结果保存到 tmpGroup 中，然后，通过 orderby 子句对 tmpGroup 按照 Key 进行排序，最后，通过 select 子句查询出来作为查询结果。

示例代码 14-12

```
class Program
{
    static void Main(string[] args)
    {
        GroupInto( );
    }
    static void GroupInto( )
    {
        //创建整数数据源
        int[] intSrc = new int[] { 1, 3, 2, 9, 5, 7, 4, 8, 6 };
        //创建查询，对数据按照模 3 的值进行分组，按照分组的 Key 进行排序
        var query = from intVal in intSrc
                    orderby intVal
                    group intVal by intVal % 3 into tmpGroup
                    orderby tmpGroup.Key
                    select tmpGroup;
        //打印查询 query 的结果
        System.Console.WriteLine("Query: ");
        //外层循环遍历所有的分组
        foreach (var grp in query)
        {
            System.Console.Write("%3={0}的元素有: ", grp.Key);
            //内层循环遍历当前分组中的所有元素
            foreach (var val in grp)
                System.Console.Write("{0}, ", val);
            System.Console.WriteLine( );
        }
    }
}
```

示例代码 14-12 的输出结果如下，与示例代码 14-11 的输出结果相比，查询结果按照分组的 Key（模 3 的值）从小到大进行排序。

```
Query:
%3=0 的元素有: 3, 6, 9,
%3=1 的元素有: 1, 4, 7,
%3=2 的元素有: 2, 5, 8,
```

技巧：理论上在一个 LINQ 中，可以出现任意数量的 select 和 group 子句，只要它们之间通过 into 子句分隔起来，只有最后一个 select 或 group 作为查询结果。但是这样会让 LINQ 可读性变差，作者建议采用多个独立的 LINQ 来表示 into 子句。

14.2.10　用并列 from 子句实现联接

在标准的 SQL 查询语言中，通常需要从多个数据源查询数据，并且数据源之间进行联接操作，在 LINQ 中有两种方式实现联接：并列 from 子句或 join 子句。并列 from 子句只是将数据源进行全联接，是简单地从多个数据源查询数据，而 join 子句则支持更多高级功能，将在 14.2.11 节详细介绍。

在 LINQ 中，通过并列 from 子句可以从多个数据源获取数据并进行查询，它的语法和单个 from 子句一样，只是每个 from 子句的临时变量名字不能相同，每个临时变量表示来自对应数据源的元素。多个 from 子句实际上可以看成是多层循环，在第 1 个 from 子句的数据源中再嵌套循环遍历第 2 个 from 子句的数据源，在第 2 个 from 子句的数据源中再嵌套循环遍历第 3 个 from 子句的数据源，依次类推。

如示例代码 14-13 所示，查询 query 中通过两个 from 子句分别从 intAry1 和 intAry2 中获取数据，前者保存在 val1 中，后者保存在 val2 中，然后打印出查询结果。

示例代码 14-13

```
static void SimpleMultiFrom( )
{
    //创建两个数据源
    int[] intAry1 = { 2, 5, 3, 10};
    int[] intAry2 = { 12, 14, 9, 13 };
    //简单从两个数据源中查询数据
    var query =
        from val1 in intAry1
        from val2 in intAry2
        select new { Val1 = val1, Val2 = val2, Sum = val1 + val2 };
    //打印查询结果
    System.Console.WriteLine("Query:");
    foreach (var item in query)
        System.Console.WriteLine(item);
}
```

示例代码 14-13 的输出如下，从中可以看出，由于这里的 from 子句并没有进行任何的数据过滤操作，所以产生的查询结果是一个完整的双层循环结果，共有 4×4=16 个元素，而且依次按照先 intAry1 后 intAry2 的次序排列。

```
Query:
{ Val1 = 2, Val2 = 12, Sum = 14 }
{ Val1 = 2, Val2 = 14, Sum = 16 }
{ Val1 = 2, Val2 = 9, Sum = 11 }
{ Val1 = 2, Val2 = 13, Sum = 15 }
{ Val1 = 5, Val2 = 12, Sum = 17 }
{ Val1 = 5, Val2 = 14, Sum = 19 }
{ Val1 = 5, Val2 = 9, Sum = 14 }
{ Val1 = 5, Val2 = 13, Sum = 18 }
{ Val1 = 3, Val2 = 12, Sum = 15 }
{ Val1 = 3, Val2 = 14, Sum = 17 }
{ Val1 = 3, Val2 = 9, Sum = 12 }
{ Val1 = 3, Val2 = 13, Sum = 16 }
{ Val1 = 10, Val2 = 12, Sum = 22 }
{ Val1 = 10, Val2 = 14, Sum = 24 }
{ Val1 = 10, Val2 = 9, Sum = 19 }
{ Val1 = 10, Val2 = 13, Sum = 23 }
```

很多开发环境中，示例代码 14-13 所示的简单的联接完全没有必要，也不符合操作需要，通常需要对来自多个数据源的元素进行某种条件的过滤操作，这就需要对数据源中的数据添加 where 子句进行过滤，而且这里的 where 子句通常是对多个数据源进行操作。

如示例代码 14-14 所示，查询 query 在示例代码 14-13 的基础上添加了 2 个 where 子句，该查询只需要数据源 intAry1 中能被 intAry2 中元素整除的元素。

示例代码 14-14

```
static void ComplexMultiFrom( )
{
    //创建两个数据源
    int[] intAry1 = { 2, 5, 3, 10 };
    int[] intAry2 = { 12, 14, 9, 13 };
    //简单从两个数据源中查询数据
    var query =
        from val1 in intAry1
        from val2 in intAry2
        where val1 < val2
        where val2 % val1 == 0
        select new { Val1 = val1, Val2 = val2, Sum = val1 + val2 };
    //打印查询结果
    System.Console.WriteLine("Query:");
    foreach (var item in query)
        System.Console.WriteLine(item);
}
```

示例代码 14-14 的输出如下，只有 val1 小于 val2，而且 val2 能被 val1 整除的两个元素作为查询结果产生，其他没有用的数据都被过滤掉。

```
Query:
{ Val1 = 2, Val2 = 12, Sum = 14 }
{ Val1 = 2, Val2 = 14, Sum = 16 }
{ Val1 = 3, Val2 = 12, Sum = 15 }
{ Val1 = 3, Val2 = 9, Sum = 12 }
```

技巧：由于多个并列的 from 子句等价于多层循环，所以效率并不高，它通常用于数据量少，而且过滤条件简单的联接查询。

14.2.11　用 join 子句实现内部联接

在 LINQ 中，还可以通过 join 子句实现联接操作。并列 from 子句只是简单的关联，而 join 子句可以将来自不同源序列并且在对象模型中没有直接关系的元素相关联，唯一的要求是每个源中的元素需要共享某个可以进行比较以判断是否相等的值。

在 LINQ 中，join 子句可以实现 3 种类型的联接：内部联接、分组联接和左外部联接。按照数据库查询定义，内部联接产生一个查询结果，对于查询结果内第一个集合中的每个元素，只要在第二个集合中存在一个匹配元算，该元素就会出现一次。如果第一个集合中的某个元素在第二个集合中没有匹配元素，则不会出现在查询结果中。

在内部联接中 join 子句的格式如下，其中 dataSource2 表示数据源，它是联接要使用的第二个数据集，element 表示存储 dataSource2 中元素的本地变量。exp1 和 exp2 表示两个表达式，它们数据类型相同，可以用 equals 进行比较，如果 exp1 和 exp2 相等，当前的元素将添加到查询结果。

```
from val1 in dataSource1
join element in dataSource2 on exp1 equals exp2
```

示例代码 14-15 中，查询 query1 从将两个数据集 intAry1 和 intAry2 联接，其中 from 子句表明联接的第一个集合为 intAry1，join 子句表明联接的第二个集合为 intAry2，on…

equals 表示在 val1 和 val2 上进行联接，当 val1%5 和 val2%15 有相同的值时，select 子句将 val1 和 val2 选择为查询结果。

示例代码 14-15

```
static void UseInnerJoin( )
{
    //创建两个整数数组 intAry1 和 intAry2 作为数据源
    int[] intAry1 = {5, 15, 25, 30, 33, 40};
    int[] intAry2 = {10, 20, 30, 50, 60, 70, 80};
    //查询 query1 使用 join 子句从两个数据源获取数据，演示内部联接的使用
    var query1 =
        from val1 in intAry1
        join val2 in intAry2 on val1%5 equals val2%15
        select new {VAL1=val1, VAL2=val2};
    //打印查询 query1 的元素
    foreach (var item in query1)
        System.Console.WriteLine(item);
}
```

示例代码 14-15 的输出如下，从中可以看出，只有 val1%5 和 val2%15 相同时，val1 和 val2 被作为查询结果，另外查询结果按照第一个集合中的元素优先排序。

```
{ VAL1 = 5, VAL2 = 30 }
{ VAL1 = 5, VAL2 = 60 }
{ VAL1 = 15, VAL2 = 30 }
{ VAL1 = 15, VAL2 = 60 }
{ VAL1 = 25, VAL2 = 30 }
{ VAL1 = 25, VAL2 = 60 }
{ VAL1 = 30, VAL2 = 30 }
{ VAL1 = 30, VAL2 = 60 }
{ VAL1 = 40, VAL2 = 30 }
{ VAL1 = 40, VAL2 = 60 }
```

14.2.12　用 join 子句实现分组联接

除了内部联接，很多开发中还需要将查询结果按照第一个数据集中的元素进行分组，这就需要使用 join 子句的另外一种用法——分组联接。分组联接的格式如下，其中 into 关键字表示将这些数据分组并保存到 grpName 中，grpName 是保存一组数据的集合。

```
from val1 in dataSource1
join element in dataSource2 on exp1 equals exp2 into grpName
```

分组联接可用于产生分层的数据结果，它将 dataSourc1 中的每个元素与 dataSource2 中的一组相关元素进行配对。值得注意的是，即使 dataSource1 中的元素在 dataSource2 中没有配对元素，也会为它产生一个空的分组对象。

在示例代码 14-16 中，使用的数据源和查询条件与示例代码 14-15 完全一样，但是它在 join 子句中添加了 into 关键字，将查询结果进行分组，然后通过嵌套的 foreach 进行遍历并显示。

示例代码 14-16

```
static void UseGroupJoin( )
{
```

```
    //创建两个整数数组 intAry1 和 intAry2 作为数据源
    int[] intAry1 = { 5, 15, 25, 30, 33, 40 };
    int[] intAry2 = { 10, 20, 30, 50, 60, 70, 80 };
    //查询 query1 使用 join 子句从两个数据源获取数据，演示分组联接的使用
    var query1 =
        from val1 in intAry1
        join val2 in intAry2 on val1 % 5 equals val2 % 15 into val2Grp
        select new { VAL1 = val1, VAL2GRP = val2Grp};
    //打印查询 query1 的元素
    foreach (var obj in query1)
    {
        System.Console.Write("{0}: ", obj.VAL1);
        foreach (var val in obj.VAL2GRP)
            System.Console.Write("{0} ", val);
        System.Console.WriteLine( );
    }
}
```

示例代码 14-16 的输出如下，从中可以看出查询结果按照 intAry1 中的元素进行分组。和示例代码 14-15 不一样，这里，intAry1 中的元素即使在 intAry2 中不存在匹配元素，也会产生一个空的列表，如 intAry1 中的元素 33。

```
5: 30 60
15: 30 60
25: 30 60
30: 30 60
33:
40: 30 60
```

14.2.13　用 join 子句实现外部联接

join 子句支持的第三种联接是左外部联接，它返回第一个集合中的所有元素，无论它是否在第二个集合中有相关元素。在 LINQ 中，通过对分组联接的结果调用 DefaultIfEmpty()执行左外部联接。DefaultIfEmpty()方法从列表中获取指定元素，如果列表为空则返回默认值。

如示例代码 14-17 所示，查询 query1 中第二个 from 子句则表示在 join 的分组结果中进行查询，查询数据源使用 DefaultIfEmpty()方法从分组结果中获取数据。

示例代码 14-17

```
static void UseLeftJoin( )
{
    //创建两个整数数组 intAry1 和 intAry2 作为数据源
    int[] intAry1 = { 5, 15, 23, 30, 33, 40 };
    int[] intAry2 = { 10, 20, 30, 50, 60, 70, 80 };
    //查询 query1 使用 join 子句从两个数据源获取数据，演示左联接的使用
    var query1 =
        from val1 in intAry1
        join val2 in intAry2 on val1 % 5 equals val2 % 15 into val2Grp
        from grp in val2Grp.DefaultIfEmpty()
        select new { VAL1 = val1, VAL2GRP = grp };
    //打印查询 query1 的元素
    foreach (var obj in query1)
        System.Console.WriteLine("{0}", obj);
}
```

示例代码 14-17 的输出如下，其中 intAry1 中的元素 23 和 33 都被列出，虽然它们在 intAry2 中不存在关联的元素。

```
{ VAL1 = 5, VAL2GRP = 30 }
{ VAL1 = 5, VAL2GRP = 60 }
{ VAL1 = 15, VAL2GRP = 30 }
{ VAL1 = 15, VAL2GRP = 60 }
{ VAL1 = 23, VAL2GRP = 0 }
{ VAL1 = 30, VAL2GRP = 30 }
{ VAL1 = 30, VAL2GRP = 60 }
{ VAL1 = 33, VAL2GRP = 0 }
{ VAL1 = 40, VAL2GRP = 30 }
{ VAL1 = 40, VAL2GRP = 60 }
```

注意： 左外部联接和分组联接虽然相似但是并非一样，分组联接返回的查询结果是一种分层数据结构，需要使用两层 foreach 才能遍历它的结果，而左外部联接是在分组联接的查询结果上再进行一次查询，所以它在 join 之后还有一个 from 子句进行查询。

14.3　使用 LINQ 方法查询

前面主要介绍 LINQ 的查询表达式及查询子句的语法，由于查询表达式语法上的限制，一些非常复杂的查询通过查询表达式很难编写，这也是传统查询语句的一个重大限制。LINQ 则不存在这一限制，它可以通过 C#代码开发的方法编写复杂的 LINQ 查询。

14.3.1　了解 Lambda 表达式和方法语法

在 LINQ 中，所有的查询、表达式、临时变量实际上都是一个对象，所以可以对这些对象进行任何支持的操作，可以访问它们的属性，调用它们的方法，甚至处理它们的事件。LINQ 查询实际是对 IEnumerable<T>对象进行操作，所有的 LINQ 查询子句都是调用该对象的某个成员（属性或方法）。如表 14-4 列出了查询子句和 IEnumerable<T>成员之间的对应关系。

表 14-4　查询子句和IEnumerable成员对应关系

IEnumerable的成员	查 询 子 句	说　　明
Cast()	指定元素类型的from子句	使用显式类型化的范围变量，例如 from int val in intAry
GroupBy()	Group……by group……by……into	对查询结果进行分组
GroupJoin()	join……in……on……equals……into	左外联接查询
Join()	join……in……on……equals	内部联接查询
OrderBy()	orderby	从小到大（顺序）排序
OrderByDescending()	orderby……descending	从大到小（逆序）排序
Select()	select	指定映射元素

续表

IEnumerable的成员	查 询 子 句	说　　明
SelectMany()	select	从多个from子句进行查询
ThenBy()	orderby……，……	多个排序元素，后一个排序元素按从小到大排序
ThenByDescending()	orderby……，……descending	多个排序元素，后一个排序元素按从大到小排序
Where()	where	条件过滤

IEnumerable<T>的成员在第 14.1.4 节已经进行介绍过，这里不再赘述。从表 14-4 中可以看出，实际上在 LINQ 查询表达式中只是使用了 IEnumerable<T>的少数方法。在实际开发中通过对查询结果或数据源进行方法调用，可以进行更多更复杂的查询操作，这也是本节的重点。

Lambda 表达式是 LINQ 方法语法的基本元素，它是一种定义匿名函数的语法，用来指定查询中需要执行的运算，通常为一些简单运算。Lambda 表达式包含表达式和语句，格式如下：

```
(parameters) => expression
```

parameters 是一个或多个输入参数，expression 是对输入参数进行运算的表达式，它们之间通过 Lambda 运算符“=>”来分隔，Lambda 表达式返回右边表达式的结果。值得注意的是，Lambda 运算符可能没有输入参数。

如示例代码 14-18 所示，查询对 IEnumerable<T>.Where()方法的调用代码，其中 Where 方法的参数为“num => num%2 == 0”，这就是一个典型 Lambda 表达式，它表示只提取能被 2 整除的数。

示例代码 14-18

```
static void LamdaExpExa( )
{
    //创建整数数组作为数据源
    int[] intAry = {2, 4, 5, 8, 9, 11};
    //通过 Lambda 表达式进行查询所有%2=0 的元素
    var query = intAry.Where(num => num%2 == 0);
    //打印查询结果
    foreach (var val in query)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

在 Lambda 表达式的定义中，parameters 表示参数列表，在 Lambda 只有一个输入参数时可以不使用括号，大于一个参数或没有参数时，括号都是必需的。两个或更多输入参数由括在括号中的逗号分隔，如下示例中，包括两个参数 p1 和 p2。

```
(p1, p2) => p1 > p2
```

通常 Lambda 表达式的参数都是可变类型的，由编译器自动确定它的具体类型。但有时编译器难于或无法推断输入类型，就需要为参数显式指定类型，即在参数之前添加参数类型。如下所示的 Lambda 表达式包括两个参数 len 和 str，其中 len 是 int 类型，而 str 则

是 string 类型。

```
(int len, string str) => str.Length > len
```

当 Lambda 表达式没有参数时，需要使用空的括号表示，如下所示，其中“( ) ”表示没有参数，而 Amethod()是一个具体的方法，该方法的返回值就是该 Lambda 表达式的结果。

```
( ) => AMethod( )
```

由于 Lambda 表达式实际是匿名函数，它可以赋值到一个委托，而在 IEnumerable<T>的方法中多数都通过函数委托来实现自定义的运算、条件等操作，所以 Lambda 表达式在 LINQ 中被广泛使用。

14.3.2　用 Where()筛选元素

在 LINQ 中，where 子句本质上是 IEnumerable<T>.Where()方法，该方法接受一个函数委托作为参数，该委托指定用来过滤元素的具体实现，它返回符合条件的元素集合。包括两个版本的 Where()方法：

（1）只对数据集合中的元素进行过滤，定义如下：

```
public static IEnumerable< TSource > Where< TSource >(
                   this IEnumerable< TSource > source,
                   Func< TSource, bool> predicate);
public delegate TResult Func<T, TResult>(T arg);
```

其中 TSource 是数据源中的元素类型，source 是被查询的数据源。predicate 是 Func<TSource, bool>类型的委托，它将数据源中的元素作为参数进行运算，并返回一个 bool 值。

（2）同时对数据集合中的元素和索引进行过滤，定义如下：

```
public static IEnumerable<TSource> Where<TSource>(
                   this IEnumerable<TSource> source,
                   Func<TSource, int, bool> predicate);
public delegate TResult Func<T1, T2, TResult>( T1 arg1, T2 arg2);
```

该版本 Where()函数的具体定义如下，参数 predicate 是 Func<TSource, int, bool>类型的委托，它将序列中的元素（TSource 类型）和元素索引分别作为第 1 个和第 2 个参数，并返回一个 bool 值。

在两个版本的 Where()方法中，都会依次对数据源中所有的元素调用 predicate 指定的委托函数，如果返回值为 true 则该元素被选择作为查询结果，否则过滤该元素。

如示例代码 14-19 所示，查询 query1 通过第 1 个版本的 Where()方法，返回集合 intSrc 中大于 15 且模 3 为 0 的元素。而查询 query2 通过第 2 个版本的 Where()方法，返回集合 intSrc 中索引小于 10 且模 3 为 0 的元素。

示例代码 14-19

```
static void Main(string[] args)
{
   SImpleWhereMethod( );
}
```

```
static void SImpleWhereMethod( )
{
    //创建整数数组作为数据源
    int[] intSrc = new int[] { 10, 14, 33, 32, 15, 27, 36, 48, 89 };
    //定义查询，只需要大于 15 并且模 3 为 0 的元素
    var query1 = intSrc.Where(num => (num > 15) && (num % 3 == 0));
    //打印查询 query1 的结果
    System.Console.Write("Query1: ");
    foreach (var item in query1)
        System.Console.Write("{0}, ", item);
    System.Console.WriteLine( );
    //定义查询，只需要索引小于 10 并且模 4 为 0 的元素
    var query2 = intSrc.Where((num, index) => (index < 10) && (num % 4 ==
    0));
    //打印查询 query2 的结果
    System.Console.Write("Query2: ");
    foreach (var item in query2)
        System.Console.Write("{0}, ", item);
}
```

示例代码 14-19 的输出如下，从中可以看出，查询 query1 和 query2 都返回了正确的查询结果，方法语法和查询语法的效果完全相同，Lambda 表达式的确带来不少方便。

```
Query1: 33, 27, 36, 48,
Query2: 32, 36, 48,
```

提示：由于 Where()方法是 IEnumerable<T>接口的扩展方法，所以像使用 IEnumerable<T>的普通成员那样使用该方法。而且，IEnumerable<T>的大部分成员都是扩展方法，定义在 Enumerable 类中。

14.3.3　用 OrderBy()对元素排序

在 LINQ 中，可以使用 OrderBy()方法从小到大（顺序）排序元素，也可以用 OrderByDescending()方法从大到小（逆序）排序元素，这两个方法各自包含两个版本，第 1 个版本只是简单给出排序表达式，定义如下：

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
                                  this IEnumerable<TSource> source,
                                  Func<TSource, TKey> keySelector)
public  static  IOrderedEnumerable<TSource>  OrderByDescending<TSource,
TKey>(
                                  this IEnumerable<TSource> source,
                                  Func<TSource, TKey> keySelector)
```

其中，TSource 是集合中元素类型，TKey 则是用于排序的值的类型。参数 keySelector 是一个 Func 类型的委托，它将元素作为参数，并返回一个要进行 TKey 类型的值，该返回值是排序的依据。

如示例代码 14-20 所示，Lambda 表达式“val=>val%10”中，val 是数据集合的元素，val%10 表示将 val 模 10 的值作为排序依据。

示例代码 14-20

```
static void UseOrderBy( )
```

```
{
    //创建 int 数组 intAry 作为数据源
    int[] intAry = { 3, -2, 5, 8, 3, 4, 2, 19, 20 };
    //查询 quer1 对 intAry 中的所有元素按照 val%10 从小到大进行排序
    var query1 = intAry.OrderBy(val => val%10);
    //打印查询 query1 的元素
    System.Console.Write("query1:");
    foreach (var val in query1)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 quer2 对 intAry 中的所有元素按照 val%10 从大到小进行排序
    var query2 = intAry.OrderByDescending(val => val%10);
    //打印查询 query2 的元素
    System.Console.Write("query2:");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

示例代码 14-20 的输出如下，从中可以看出，查询 query1 将 intAry 中的元素按照模 10 的值从小到大排序，查询 query2 将 intAry 中的元素按照模 10 的值从大到小排序。

```
query1:-2 20 2 3 3 4 5 8 19
query2:19 8 5 4 3 3 2 20 -2
```

在 OrderBy()和 OrderByDescending()方法中，如果没有指定数据比较器，则使用默认的数据比较器，示例代码 14-20 使用默认的 int 比较器，所以负数小于 0。在一些开发中，需要使用特定的数据比较器，数据比较器需要实现 IComparer<TKey>接口，OrderBy()和 OrderByDescending()方法都提供了一个接受特定数据比较器的方法接口，定义如下：

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
                                        this IEnumerable<TSource> source,
                                        Func<TSource, TKey> keySelector,
                                        IComparer<TKey> comparer)
public  static  IOrderedEnumerable<TSource>  OrderByDescending<TSource,
TKey>(
                                        this IEnumerable<TSource> source,
                                        Func<TSource, TKey> keySelector,
                                        IComparer<TKey> comparer)
```

其中，参数 comparer 是 IComparer<TKey>类型的对象，它提供自定义的数据比较器。如示例代码 14-21 中，其中 MyComparer 类实现接口 IComparer<int>，它实现 Compare()方法支持对 int 类型数据比较，实际上是比较 int 数的绝对值。

示例代码 14-21

```
//自定义的 int 类型比较器，实现 IComparer<int>接口
class MyComparer : IComparer<int>
{
    //比较函数具体实现，对 x 和 y 的绝对值进行比较
    public int Compare(int x, int y)
    {
        int x1 = Math.Abs(x);
        int y1 = Math.Abs(y);
        //如果|x|>|y|，返回 1
        if (x1 > y1) return 1;
```

```
        //如果|x|=|y|，返回 0
        else if(x1 == y1) return 0;
        //如果|x|<|y|，返回-1
        else return -1;
    }
}
static void UseOrderByDef( )
{
    //创建自定义 int 类型比较器 MyComparer 对象 mc
    MyComparer mc = new MyComparer( );
    //创建 int 数组 intAry 作为数据源
    int[] intAry = { 3, -2, 5, 8, -3, -4, 2, -19, 20 };
    //查询 query1 对 intAry 中的所有元素，使用自定义比较器从小到大排序
    var query1 = intAry.OrderBy(val => val, mc);
    //打印查询 query1 的元素
    System.Console.Write("query1:");
    foreach (var val in query1)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query2 对 intAry 中的所有元素，使用自定义比较器从大到小排序
    var query2 = intAry.OrderByDescending(val => val, mc);
    //打印查询 query2 的元素
    System.Console.Write("query2:");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
   System.Console.WriteLine( );
}
```

示例代码 14-21 的输出如下，从中可以看出，query1 和 query2 分别对 intAry 中的元素根据绝对值按照从小到大及从大到小的顺序进行排序。

```
query1:-2 2 3 -3 -4 5 8 -19 20
query2:20 -19 8 5 -4 3 -3 -2 2
```

提示： IEnumberable<T>的成员中还包括很多和查询子句等价的方法，它们的使用和Where()相似，这里不再赘述，有兴趣的读者可以参看 MSDN 或相关书籍。

14.4　对 LINQ 查询结果执行集合操作

在 LINQ 中，通过 IEnumerable<T>类，提供了很多集合类操作，比如求和、最大值等，也包括提取指定条件的元素。集合的连接、并集、交集等。本节将介绍其中比较典型的几个方法。

14.4.1　用 Average()等执行数值计算

传统 SQL 查询语言支持对数据表中的数据进行某些特定的数值运算，包括求最大值、求最小值、求平均值、求和。同样地，IEnumerable<T>也提供了等价方法完成这些运算，如下所示。

- Min()：计算集合中指定元素的最小值。

- ❑ Max()：计算集合中指定元素的最大值。
- ❑ Sum()：计算集合中指定元素的累加和。
- ❑ Average()：计算集合中指定元素的平均值。

这些数值计算函数，都包括 13 个重载版本，最简单的一个版本不接受任何参数，此时参与计算的元素类型必须具有默认的数值运算支持。其中，Max()和 Min()都需要类型具有排序功能，这些类型包括 int、float 等数值类型，string 字符串类型，枚举类型等。而 Sum()和 Average()两个方法，则需要目标类型能够默认转化成 int、float 等数值类型，从而进行累加和平均计算。

示例代码 14-22 演示 Min()、Max()、Sum()、Average()方法的使用，其中前半段代码计算 intAry 集合中的最大值、最小值、和、平均值。后半段代码计算 strAry 的最大值和最小值。

示例代码 14-22

```
static void UseValueCalc( )
{
    int[] intAry = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
                                             //创建 int 类型数组 intAry 作为数据源
    var intMax = intAry.Max( );              //intAry 中的最大值
    var intMin = intAry.Min( );              //intAry 中的最小值
    var intAverage = intAry.Average( );      //intAry 中元素的平均值
    var intSum = intAry.Sum( );              //intAry 中元素的累加和
    //打印计算结果
    System.Console.WriteLine("intAry's max = {0}, min = {1}, average = {2},
    sum = {3}",
                                  intMax, intMin, intAverage, intSum);
    //创建 string 数组 strAry 作为数据源
    string[] strAry = {"Hello", "hello", "thanks", "alibaba", "street" };
    var strMax = strAry.Max( );              //strAry 中的最大值
    var strMin = strAry.Min( );              //strAry 中的最小值
    //打印计算结果
    System.Console.WriteLine("strAry's max = {0}, min = {1}", strMax,
    strMin);
}
```

示例代码 14-22 的输出如下，从中可以看出虽然 intAry 的所有元素都为 int 类型，但平均值 intAverage 会自动变化成 float 类型，从而可以保留小数部分。字符串大小是按照字母表的顺序进行比较。

```
intAry's max = 10, min = 1, average = 5.5, sum = 55
strAry's max = thanks, min = alibaba
```

注意： 由于 string 类型并没有提供直接转化到数值类型（int、float 等）的方法，所以不能对 strAry 进行求和或平均操作，即代码 strAry.Sum()是错误的语法。

实际上，在很多开发应用中，需要对非数值类型的数据进行求和、求平均、求最大值、求最小值等数值操作，这就需要是用数值操作的重载版本。在这些重载版本中，接受一个函数委托类型的参数，该委托将特定类型的数据转换成数值类型，从而进行累加等操作。

在示例代码 14-23 中，通过 Lambda 表达式“str=>str.Length”计算字符串长度。所以

可以对 strAry 使用 Sum()、Average()等操作。其中 strMax 是所有字符串长度的最大值，strMin 是所有字符串长度最小值，strSum 是所有字符串长度的和，strAverage 是所有字符串长度的平均值。

示例代码 14-23

```
static void UseValueCalc2( )
{
    //创建 string 数组 strAry 作为数据源
    string[] strAry = { "Hello", "hel", "thanks", "alibaba", "street" };
    var strMax = strAry.Max(str => str.Length);
                                        //strAry 中的元素字符串长度最大值
    var strMin = strAry.Min(str => str.Length);
                                        //strAry 中的元素字符串长度最小值
    //打印计算结果
    System.Console.WriteLine("strAry's max = {0}, min = {1}", strMax,
    strMin);
    var strSum = strAry.Sum(str => str.Length);
                                        //strAry 中的元素字符串长度累加和
    var strAverage = strAry.Average(str => str.Length);
                                        //strAry 中的元素字符串长度平均值
    //打印计算结果
    System.Console.WriteLine("strAry's sum = {0}, average = {1}", strSum,
    strAverage);
}
```

示例代码 14-23 的输出如下，其中 Max()、Min()操作由于使用了参数，所以输出结果和 14-22 完全不一样。另外，通过 strSum 和 strAverage 可以看出，使用 Lambda 表达式使得字符串转化成数值，并进行运算。

```
strAry's max = 7, min = 3
strAry's sum = 27, average = 5.4
```

技巧：实际开发中，可以按照某种特定规则将非数值类型数据映射到数值类型，然后使用 Min()、Max()等操作获取最大值、最小值等，再根据返回值反推出该数据。

14.4.2　用 Skip()和 SkipWhile()跳过元素

在一些开发场景中，需要跳过某些元素，只提取剩下的元素作为查询结果，这就需要使用 IEnumerable<T>接口的 Skip()或 SkipWhile()两个方法。它们都是用来跳过集合中的元素，Skip()只是简单地跳过集合中指定数量的元素，而 SkipWhile()则跳过集合中满足指定条件的元素，定义如下：

```
public static IEnumerable<TSource> Skip<TSource>(
                            this IEnumerable<TSource> source,
                            int count)
public static IEnumerable<TSource> SkipWhile<TSource>(
                            this IEnumerable<TSource> source,
                            Func<TSource, bool> predicate)
public static IEnumerable<TSource> SkipWhile<TSource>(
                            this IEnumerable<TSource> source,
                            Func<TSource, int, bool> predicate)
```

其中，参数 count 在 Skip()中使用，指定要跳过的元素数量，被跳过的元素从集合中第 1 个元素开始计数。参数 predicate 是 Func 类型委托，在 SkipWhile()中指定要跳过的元素所满足的条件，它的第 1 个参数是集合中的元素值，第 2 个参数是该元素的索引。

SkipWhile()从集合中第 1 个元素开始，使用参数 predicate 进行计算，如果返回 true，跳过并继续判断下一个元素；如果 predicate 返回 false，则停止判断，返回集合中所有没有被跳过的元素。

示例代码 14-24 演示 Skip()和 SkipWhile()的使用，查询 query1 跳过集合中前 3 个元素，query2 则跳过集合中绝对值小于 10 的元素。

示例代码 14-24

```
static void UseSkip( )
{
    //创建 int 类型数组 intAry 作为数据源
    int[] intAry = { 3, -2, 5, 8, -13, -4, 12, -19, 20 };
    //查询 query1 跳过 intAry 中的前 3 个元素
    var query1 = intAry.Skip(3);
    //打印查询 query1 的元素
    System.Console.Write("query1: ");
    foreach (var val in query1)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query2 跳过 intAry 中，从第 0 个元素开始连续的绝对值小于 10 的元素
    var query2 = intAry.SkipWhile(num => num / 10 == 0);
    //打印查询 query2 的元素
    System.Console.Write("query2: ");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

示例代码 14-24 的输出如下，其中 query1 跳过前面 3 个元素，query2 则跳过所有绝对值小于 10 的元素，即前面 4 个元素。虽然 intAry 的第 6 个元素绝对值也小于 10，但是由于第 5 个元素终止了 SkipWhile()的循环判断，所以第 5 个及后面的元素都不会跳过。

```
query1: 8 -13 -4 12 -19 20
query2: -13 -4 12 -19 20
```

注意： Skip()和 Where()的区别在于，Where()会对数据源中的所有元素都进行过滤，而 Skip()和 SkipWhile()并不是过滤所有元素，它在遇到不满足条件的元素后，剩下的元素不再进行判断。

14.4.3　用 Take()和 TakeWhile()提取元素

和 Skip()相反，IEnumerable<T>还提供了 Take()和 TakeWhile()方法，从集合的开头提取满足指定条件的元素，这两个函数的定义和 Skip()类似，如下所示。

```
public static IEnumerable<TSource> Take<TSource>(
                                    this IEnumerable<TSource> source,
                                    int count)
```

```
public static IEnumerable<TSource> TakeWhile<TSource>(
                                this IEnumerable<TSource> source,
                                Func<TSource, bool> predicate)
public static IEnumerable<TSource> TakeWhile<TSource>(
                                this IEnumerable<TSource> source,
                                Func<TSource, int, bool> predicate)
```

其中，参数 count 在 Take()中使用，指定要提取的元素数量，被提取的元素从集合中第 1 个元素开始计数。参数 predicate 是 Func 类型委托，在 TakeWhile()中指定要提取元素所满足的条件，它的第 1 个参数是集合中的元素值，第 2 个参数是该元素的索引。

TakeWhile()从集合中第 1 个元素开始，使用参数 predicate 进行计算，如果返回 true，提取并继续判断下一个元素，如果 predicate 返回 false，则停止判断，返回已经提取的元素。

示例代码 14-25 演示 Take()和 TakeWhile()的使用，查询 query1 提取集合中的前 3 个元素，query2 则提取集合中绝对值小于 10 的元素。

示例代码 14-25

```
static void UseTake( )
{
    //创建 int 类型数组 intAry 作为数据源
    int[] intAry = { 3, -2, 5, 8, -13, -4, 12, -19, 20 };
    //查询 query1 提取 intAry 中的前 3 个元素
    var query1 = intAry.Take(3);
    //打印查询 query1 的元素
    System.Console.Write("query1: ");
    foreach (var val in query1)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query2 提取 intAry 中，从第 0 个元素开始连续的绝对值小于 10 的元素
    var query2 = intAry.TakeWhile(num => num / 10 == 0);
    //打印查询 query2 的元素
    System.Console.Write("query2: ");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

示例代码 14-25 的输出如下，其中 query1 跳过前 3 个元素，query2 则跳过所有绝对值小于 10 的元素，即前 4 个元素。示例代码 7-26 中，虽然 intAry 的第 6 个元素绝对值也小于 10，但是由于第 5 个元素终止了 TakeWhile()的循环判断，所以第 5 个及后面的元素都不会被提取。

```
query1: 3 -2 5
query2: 3 -2 5 8
```

14.4.4　用 Distinct()方法消除相等元素

在一些场合下，一个数据集合通常包含很多条记录，并且这些记录存在重复的情况，通常需要对这些记录进行唯一性判断，即重复出现的记录只保留一次。在 LINQ 中，可以通过 IEnumerable<T>提供的 Distinct()方法完成，Distinct()方法包括一个不带参数的版本，

它使用默认的相等比较器对集合中的元素进行比较，定义如下：

```
public static IEnumerable<TSource> Distinct<TSource>(
                                this IEnumerable<TSource> source)
```

对于简单的数据类型，比如 int、float、short、double、string 等，通过默认的相等比较器就可以很好地解决重复数据问题。如示例代码 14-26 中，intAry 是一个 int 数组，它包含多个相等元素，查询 query1 通过 Distinct()方法消除所有重复的元素。

示例代码 14-26

```
static void UseDistinctSimple( )
{
    //创建 int 类型数组 intAry 作为数据源
    int[] intAry = {1, 3, 6, 9, 1, 3, 9, 10, 2, 5, 3, 10};
    //查询 query1 对 intAry 中的元素进行消除重复操作
    var query1 = intAry.Distinct( );
    //打印查询 query1 的元素
    System.Console.Write("query1: ");
    foreach (var item in query1)
    {
        System.Console.Write("{0} ", item);
    }
}
```

示例代码 14-26 的输出如下，可以看出相等的元素只出现了一次，并且按照第一次出现的先后顺序保存在查询结果中。

```
query1: 1 3 6 9 10 2 5
```

当元素不是 int 等简单类型，甚至该类型根本不存在默认的相等比较器时，要对集合进行消除重复操作就需要使用 Distinct()的另外一个版本，该版本需要指定使用者提供一个相等比较器，Distinct()方法通过该比较器进行元素重复判断。定义如下：

```
public static IEnumerable<TSource> Distinct<TSource>(
                                 this IEnumerable<TSource> source,
                                 IEqualityComparer<TSource> comparer)
```

其中，参数 comparer 是 IEqualityComparer<T>类型对象，定义如何进行元素重复判断，通常是开发人员自己编写的实现 IEqualityComparer<T>接口的类对象。

IEqualityComparer<T>接口包括两个成员方法：Equals()和 GetHashCode()，它们的定义如下：

```
bool Equals(T x, T y)
int GetHashCode(T obj)
```

其中，T 是具体的元素类型。Distinct()方法按照图 14-3 所示的流程图进行元素重复性判断。

从图 14-3 中可以看出，开发人员根据实际开发需要实现 GetHashCode()和 Equals()方法才能充分进行自定义的重复性判断。而元素的哈希码和相等方法作为双重条件对元素进行判断，只有两个元素的哈希码相等，而且在 Equals()方法中也判断为相等时才算是重复元素。

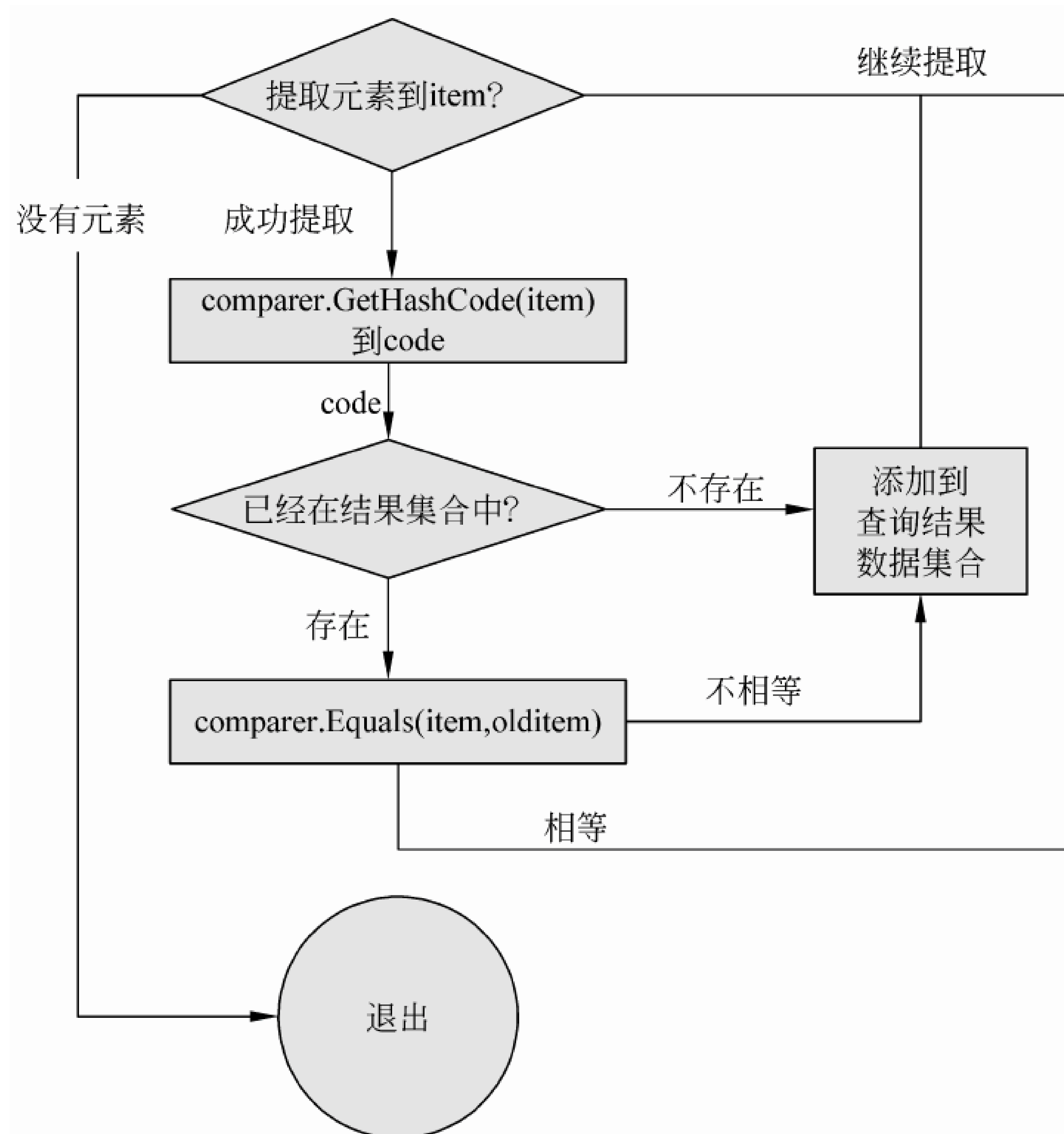


图 14-3　Distinct()方法执行流程图

在示例代码 14-27 中，MyStrEqualComparer 类是自定义的字符串元素相等比较器，所以它实现接口 IEqualityComparer<string>。在 GetHashCode()中将字符串长度作为字符串的长度，在 Equals()中如果两个元素的第一个字符相等的，则认为是相等的。查询 query2 则使用 MyStrEqualComparer 类，对字符串集合 strAry 中的元素进行消除重复元素的操作。

示例代码 14-27

```
//自定义的字符串相等比较器，实现 IEqualityComparer<string>接口
class MyStrEqualComparer : IEqualityComparer<string>
{
    //实现 Equals()方法
    bool IEqualityComparer<string>.Equals(string x, string y)
    {
        //如果两个字符中的第一个字符相等，则返回 true
        return x.Substring(0,1) == y.Substring(0,1);
    }
    //实现 GetHashCode()方法
    public int GetHashCode(string obj)
    {
        //将字符串的长度作为它的哈希码
        return obj.Length;
    }
}
static void UseDistinctComplex( )
{
    //创建 string 类型数组 strAry 作为数据源
    string[] strAry = {"hel", "how", "hello", "tower", "his", "aim", "aha",
    "hints"};
    //创建自定义 string 相等比较器 MyStrEqualComparer 对象 mcStr
```

```
    MyStrEqualComparer mcStr = new MyStrEqualComparer();
    //查询 query2 使用自定义相等比较器 mcStr 对 strAry 执行消除重复操作
    var query2 = strAry.Distinct(mcStr);
    //打印查询 query2 的元素
    System.Console.Write("query2: ");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

示例代码 14-27 的输出如下，how 和 hel 具有相同的字符串长度和相同的首字符，所以认为是重复的，就只有第一个字符串 how 被作为查询结果。hello 和 hel 虽然具有相同的首字符，但是由于长度不等，所以认为不是重复的。

```
query2: hel hello tower aim
```

技巧： 在编写自定义的相等比较器时，将 HashCode 和 Equals()看成是两个具有主次关系的条件进行区分，前者为主，后者为次，这样可以使比较器的设计和实现变得更加清晰，功能更明确。

14.4.5　用 Concat()连接两个集合

在 LINQ 中，还可以通过 IEnumerable<T>的 Concat()方法将两个集合中的元素首尾相连，从而构成一个新的 IEnumerable<T>对象。Concat()方法定义如下：

```
public static IEnumerable<TSource> Concat<TSource>(
                               this IEnumerable<TSource> first,
                               IEnumerable<TSource> second)
```

其中，second 参数是被连接的数据集合，两个数据集合中的元素必须是相同类型，否则不能进行连接操作。示例代码 14-28 演示 Concat()方法的使用，其中 query1 和 query2 是将集合 strAry1 和 strAry2 中的元素连接，两者连接顺序相反。

示例代码 14-28

```
static void UseContact( )
{
    //创建两个 string 数组 strAry1 和 strAry2 作为数据源
    string[] strAry1 = {"Hello,", "Nice", "to", "meet", "you!",};
    string[] strAry2 = {"Jone", "Smith", "Phil" };
    //查询 query1 将 strAry2 连接到 strAry1 之后
    var query1 = strAry1.Concat(strAry2);
    //打印查询 query1 的元素
    System.Console.Write("query1: ");
    foreach (var item in query1)
        System.Console.Write("{0} ", item);
    System.Console.WriteLine( );
    //查询 query2 将 strAry1 连接到 strAry2 之后
    var query2 = strAry2.Concat(strAry1);
    //打印查询 query2 的元素
    System.Console.Write("query2: ");
    foreach (var item in query2)
        System.Console.Write("{0} ", item);
```

```
    System.Console.WriteLine( );
}
```

示例代码 14-28 的输出如下，可以看出 query1 是 strAry1 的元素连接 strAry2 的元素，query2 是 strAry2 的元素连接 strAry1 的元素。

```
query1: Hello, Nice to meet you! Jone Smith Phil
query2: Jone Smith Phil Hello, Nice to meet you!
```

注意：Concat()方法是直接将两个集合中的元素连接在一起，不会进行重新排序、过滤等，就算两个集合中的元素有重复现象也同样保留。

14.4.6　用 Intersect()等集合操作

集合的并集、交集、差集等，是集合的常用操作，在 LINQ 中，IEnumerable<T>类分别通过 Union()、Intersect()、Except()完成这 3 个操作。这 3 个方法各自都包含 2 个重载版本，1 个版本不需要参数，它是默认相等比较器进行元素相等比较。

（1）Union()：该方法对集合 A 和集合 B 进行并集操作，返回两个集合中的所有元素，相等的元素只出现一次，定义如下：

```
public static IEnumerable<TSource> Union<TSource>(
                              this IEnumerable<TSource> first,
                              IEnumerable<TSource> second)
```

（2）Intersect()：该方法对集合 A 和集合 B 进行交集操作，返回两个集合中的相等元素，定义如下：

```
public static IEnumerable<TSource> Intersect<TSource>(
                              this IEnumerable<TSource> first,
                              IEnumerable<TSource> second)
```

（3）Except()：该方法对集合 A 和集合 B 进行差集操作，返回在集合 A 中有，但是集合 B 中没有的元素，定义如下：

```
public static IEnumerable<TSource> Except<TSource>(
                              this IEnumerable<TSource> first,
                              IEnumerable<TSource> second)
```

该简单版本的方法使用默认相等比较器，所以通常对集合元素类型为 int、string 等值类型使用。而且对于同样的两个集合 A 和 B，A.Union(B)和 B.Union(A)返回集合中包含相同的元素。A.Intersect(B)和 B.Intersect(A)返回的集合中也包含相同的元素。但 A.Except(B)和 B.Except(A)返回集合的元素则不相同，前者集合中元素来自集合 A，后者元素来自集合 B。

如示例代码 14-29 所示，其中 intAry1 和 intAry2 是两个 int 类型数据集合。查询 query1 计算 intAry1 和 intAry2 的并集，查询 query2 计算 intAry1 和 intAry2 的交集。查询 query3 计算 intAry1 和 intAry2 的差集，查询 query4 计算 intAry2 和 intAry1 的差集。

示例代码 14-29

```
static void UseSetOpSimple( )
```

```
{
    //创建两个 int 类型数组 intAry1 和 intAry2 作为数据源
    int[] intAry1 = {1, 3, 5, 8, 10, 12, 33, 45, 12, 2};
    int[] intAry2 = {2, 5, 10, 6, 7, 11, 23, 25, 45, 33};
    //查询 query1 返回 intAry1 和 intAry2 的并集
    var query1 = intAry1.Union(intAry2);
    //打印查询 query1 的元素
    System.Console.Write("query1: ");
    foreach (var val in query1)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query2 返回 intAry1 和 intAry2 的交集
    var query2 = intAry1.Intersect(intAry2);
    //打印查询 query2 的元素
    System.Console.Write("query2: ");
    foreach (var val in query2)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query3 返回 intAry1 对 intAry2 的差集
    var query3 = intAry1.Except(intAry2);
    //打印查询 query3 的元素
    System.Console.Write("query3: ");
    foreach (var val in query3)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
    //查询 query4 返回 intAry2 对 intAry1 的差集
    var query4 = intAry2.Except(intAry1);
    //打印查询 query4 的元素
    System.Console.Write("query4: ");
    foreach (var val in query4)
        System.Console.Write("{0} ", val);
    System.Console.WriteLine( );
}
```

示例代码 7-32 的输出如下，其中 query1 返回集合中包含了 intAry1 和 intAry2 中的所有元素，元素按照先 intAry1 后 intAry2 的顺序排列。查询 query2 返回集合中 intAry1 和 intAry2 都有的元素。查询 query3 返回在集合 intAry1 中有但 intAry2 中没有的元素集合。query4 返回在集合 intAry2 中有但 intAry1 中没有的元素集合。

```
query1: 1 3 5 8 10 12 33 45 2 6 7 11 23 25
query2: 5 10 33 45 2
query3: 1 3 8 12
query4: 6 7 11 23 25
```

此外，Union()、Intersect()、Except()除了上面的基本版本之外，还包括一个需要 IEqualityComparer<T>类型参数的版本。通过这些函数，开发人员可以为特定的类提供自定义的相等比较器，从而满足特殊的应用需求。关于 IEqualityComparer<T>的使用，读者可以参考 14.4.4 节，这里不再赘述。

14.5　小　　结

LINQ 在.NET 4.0 中提供支持，将数据查询与编程语言很好集成到一起，使得编写数

据查询操作更加快速和轻松。

LINQ 查询是.NET 4.0 的一个重要特性，它与 C#良好集成，通过 IEnumerable<T>接口提供对数据集合进行提取、查询、数值运算、连接、联接、并集、交集、差集等操作。通过这些操作可以对数据集合进行各种操作，但是这些操作都是只读的，原集合本身并不会发生变化。

本章通过众多实例详细介绍了 LINQ 对内存数据查询的具体内容，通过本章的学习，读者应该掌握以下知识点：

- 什么是 LINQ？
- LINQ 的相关组件有哪些？
- 如何在 C#中使用 LINQ？
- 什么是 LINQ 查询表达式？
- from 子句有什么用？并列 from 子句如何使用？
- select 子句如何使用？
- where 子句如何使用？如何实现多个 where 子句？
- orderby 子句如何使用？如何实现多个关键字排序？
- group 子句分组如何实现？
- join 子句如何使用？
- IEnumerable<T>接口及其成员有什么作用？
- Lambda 表达式如何使用？
- IEnumerable<T>接口成员和 LINQ 查询表达式的关系如何？
- 如何在集合中进行数据提取？
- 如何在集合中进行数值运算？
- 如何对多个集合进行集合操作？

第 15 章　LINQ to DataSet

ADO.NET 是.NET Framework 与数据库交互的关键组件，它提供统一的接口访问多种流行的数据库。在.NET 4.0 中，将 LINQ 与 ADO.NET 结合，充分利用 LINQ 查询高效、类型安全等特点，提供更加丰富和高效的数据库查询操作，本章将详细介绍这一技术的分支——LINQ to DataSet。

15.1　了解 LINQ to DataSet

LINQ to ADO.NET 是.NET 4.0 的重要技术，它包括 LINQ to DataSet 和 LINQ to SQL 两个分支，前者支持对内存中 DataSet、DataTable 进行查询，后者支持从数据库获取数据并进行数据查询。本节简单介绍 LINQ to DataSet 的特点。

15.1.1　了解 LINQ to ADO.NET

随着数据库应用的日益广泛，现在的开发人员通常需要掌握至少两类开发语言：用于业务逻辑开发的高级语言（如 C#或）和数据库开发查询语言（如 Transact-SQL）。在开发应用软件时，业务开发人员和数据库开发人员往往需要不断的交流和沟通，这样既影响开发效率，也难以维护。

LINQ to ADO.NET 是.NET 4.0 在推出 LINQ 之后对 ADO.NET 的增强，它将 LINQ 和 ADO.NET 紧密结合，充分利用 LINQ 强大的对象数据查询能力和 ADO.NET 完善的多数据库数据操作能力，使得开发人员可以在脱离数据库开发人员的基础上进行数据库查询及业务逻辑的开发。

如图 15-1 所示，目前有 3 种独立的 LINQ to ADO.NET 技术：LINQ to DataSet、LINQ to SQL 和 LINQ to Entities。LINQ to DataSet 能够利用 LINQ 对内存数据集 DataSet 执行形式多样的优化查询，LINQ to SQL 可以直接查询 SQL Server 数据库架构，而 LINQ to Entities 可以查询实体数据模型。其中，前面两种是常用的技术，也是本章和第 16 章要详细介绍的技术。

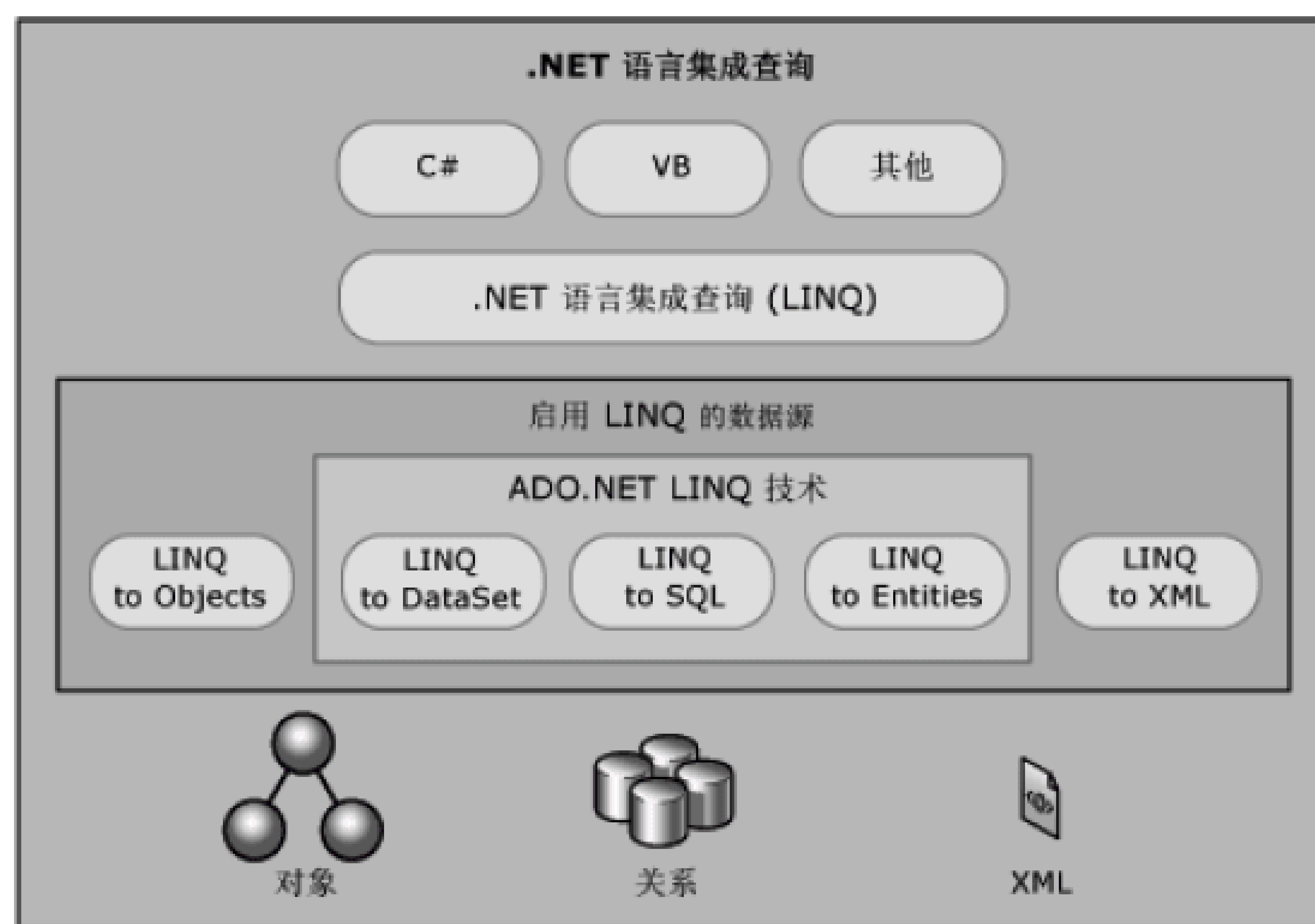


图 15-1　LINQ to ADO.NET 结构

通过 SQL 语句将数据从数据库传输到内存对象中，通常单调乏味而且容易出错。在 LINQ to ADO.NET 中，由 LINQ to DataSet

和 LINQ to SQL 实现的 LINQ 提供程序可以将源数据转换为基于 Ienumerable<T>的对象集合。这样业务开发人员可以通过 LINQ 对数据库中的数据进行更多功能更强大的查询功能，而且利用 Visual Studio 2010 的编译器和智能感知特性，可以极大地提高开发效率。

通过 LINQ to ADO.NET，业务开发人员可以将精力完全集中在业务逻辑上，同时也可以将一些和数据关系不大的高级查询操作，从数据库服务器转移到业务层。而数据库开发人员则重点在于为业务层提供完整而不冗余的数据集合，不用花心思在更多十分复杂的 SQL 查询上。

15.1.2　了解 LINQ to DataSet

在 ADO.NET 中，将数据库中的数据以数据集合（DataSet）和数据表（DataTable）的形式保存在内存中，而 DataSet 可以简单看成是多个 DataTable 的集合，当然还附带一些数据管理功能。DataSet 是 ADO.NET 进行无链接模式数据库访问的关键元素，在 UI 层，DataSet 与界面控件集成并进行数据绑定。在中间层上，DataSet 提供保存数据关系的缓存并包括快速简单查询和层次结构导航服务。

DataSet 的另一个有用特征是允许应用程序将数据子集从一个或多个数据源导入应用程序空间，从而应用程序可以在内存中操作这些数据，同时还保留其关系。DataSet 在数据管理上有着突出的优点，但是它对数据查询功能的支持却有限，尤其是对于复杂的情况，开发人员必须在数据库服务器端编写复杂的自定义查询来获取特定数据，这样大大降低了应用程序的扩展性，也加大了数据流量。

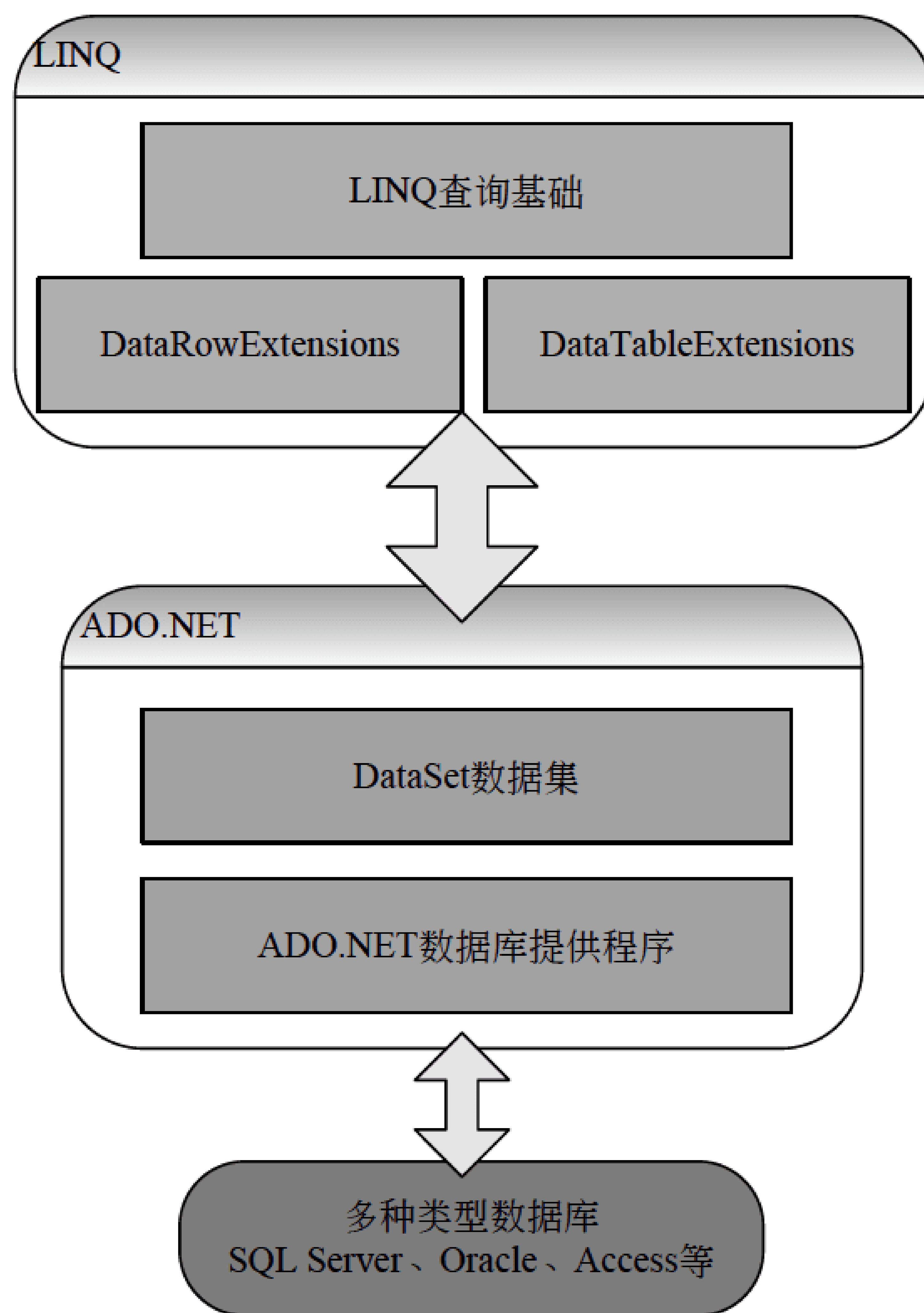


图 15-2　LINQ to DataSet 与 ADO.NET

在 LINQ to DataSet 中，将 LINQ 和 ADO.NET 完整集成，如图 15-2 所示，ADO.NET 负责从数据库中将数据读入到 DataSet（或 DataTable）中，LINQ to DataSet 则查询在 DataSet 和 DataTable 对象中缓存的数据。很显然，由于 ADO.NET 支持多种数据库，所以就实现了通过 LINQ 来间接查询多种数据库中的数据。

只有在填充 DataSet 后，才能使用 LINQ to DataSet 查询 DataSet 对象。LINQ to DataSet 通过 DataRowExtensions 和 DataTableExtensions 类中的扩展方法，更快更容易地查询 DataSet 中的数据。LINQ 查询可以对 DataSet 中的单个表执行，也可以通过使用 Join 和 GroupJoin 标准查询运算符对多个表执行。

开发人员能够使用编程语言本身而不是使用单独的查询语言来编写查询，所以 LINQ to DataSet 大大简化了查询代码的编写。同时，可以在 LINQ to DataSet 中利用 Visual Studio

所提供的编译时语法检查、静态类型和智能感知支持的功能，使得代码编写更加快速和安全。

LINQ to DataSet 本质上来说还是在内存中对数据进行查询和操作，它只是借助于 ADO.NET 实现数据库数据读取和支持。所以它的重点是两个重要的静态类：DataRow Extensions 和 DataTableExtensions。这两个类为 LINQ 和 DataSet 提供联接的桥梁，DataTable Extensions 提供的扩展方法将 DataTable 转化成 LINQ 可以查询的 IEnumerable<T>类型，DataRowExtensions 提供的扩展方法支持对 DataRow 字段（Field）数据的类型化读取和设置，表 15-1 和表 15-2 分别给出了它们的成员。在 15.2 节将详细介绍 LINQ to DataSet 的具体应用。

表 15-1　DataTableExtensions成员

成　　员	功　　能
AsDataView	创建并返回支持 LINQ 的 DataView 对象
AsEnumerable	返回一个 IEnumerable<DataRow>对象，此对象可用在 LINQ 表达式或方法查询中
CopyToDataTable	在给定输入 IEnumerable<T>对象的情况下，返回包含 DataRow 对象副本的 DataTable

表 15-2　DataRowExtensions成员

成　　员	功　　能
Field	提供对 DataRow 中的每个列值的强类型访问
SetField	为 DataRow 中的指定列设置一个新值

15.2　使用 LINQ to DataSet 查询数据

LINQ to DataSet 将 LINQ 和 ADO.NET 集成，通过 ADO.NET 将数据读取到 DataSet/DataTable 中，通过 LINQ 对 DataSet/DataTable 进行查询，从而实现对数据库数据的复杂查询。本节将介绍如何使用 LINQ to DataSet 进行数据查询。

15.2.1　LINQ to DataSet 开发步骤

数据源是 LINQ 查询的核心元素，在 LINQ to DataSet 中，DataSet 和 DataTable 就为 LINQ 查询提供了类似于数据库的数据源。可以简单地将 LINQ to DataSet 看成是通过 LINQ 对 DataSet 中保存的数据进行查询，它和第 13 章介绍的 LINQ 查询没有本质区别，语法也类似。

在使用 LINQ to DataSet 时，通常需要以下 4 个步骤，其中第 2 步是普通的 LINQ 查询不需要的，第 1 步和第 2 步创建了查询的数据源：

（1）获取 DataSet/DataTable 数据源。

先要准备 DataSet/DataTable 数据源，可以通过 ADO.NET 技术从数据库获取，可以通过 XML 技术从 XML 文件获取，也可以从其他任何形式的数据源获取，甚至可以在内存中直接创建并填充 DataSet/DataTable 对象。

（2）将 DataTable 转换成 IEnumerable<T>类型。

LINQ 只能在 IEnumerable<T>或 IQueryable<T>接口对象上执行查询操作，而 DataTable 并没有实现这两个接口，所以不能直接查询 DataTable 对象。在 LINQ to DataSet 中，通过 DataTableExtensions 扩展的 AsEnumerable() 方法从 DataTable 获取一个等价的 IEnumerable<T>对象。

（3）使用 LINQ 语法编写查询。

LINQ to DataSet 中查询的编写可以使用查询语法和方法语法，可以对它执行任何 IEnumerable<T>允许的查询操作，包括过滤、排序、联接等。

（4）使用查询结果。

LINQ 查询执行完成后，就可以使用查询结果，比如用 foreach 遍历所有元素，用 Max() 等进行数值计算，将它作为数据源进行二次查询等。

本章将通过实例详细介绍 LINQ to DataSet 的具体使用，为了更容易理解，部分示例中的 DataSet 通过代码直接在内存中编写，并不从数据库获取。

注意：由于 DataSet 本身是 DataTable 的集合，它可以包含一个或多个 DataTable 及它们之间的关系，LINQ to DataSet 实际是对 DataTable 进行数据查询，并非对 DataSet 进行查询。

15.2.2　查询单个 DataTable 的记录

DataSet 可以看成是一个内存数据库，它包含一个或多个 DataTable，同时也维护 DataTable 之间的约束和关系。LINQ to DataSet 更多是对 DataSet 中的 DataTable 进行查询，因为 DataTable 才是真正保存数据的地方。被查询的 DataTable 可以来自单个 DataSet，也可以是来自多个 DataSet，甚至可以是一个不属于任何 DataSet 的独立的 DataTable。

在 15.2.1 节介绍了查询 DataTable 中元素的主要步骤，在对 DataTable 进行数据查询时必须使用 DataTable 的扩展方法 AsEnumerable()。该方法将 DataTable 转换成类型为 IEnumerable<DataRow>的可枚举数据集合，它的定义如下：

```
public static EnumerableRowCollection<DataRow> AsEnumerable(
                     this DataTable source)
```

因此，从 DataTable 中获取的元素类型为 DataRow。要进一步访问数据表记录的具体字段数据，就需要使用 DataRow 的泛型扩展方法 Field<T>，通过它获取 DataRow 的某字段的数据，包括 6 个重载版本，其中最常用的有下面 3 个。

```
public static T Field<T>( this DataRow row, DataColumn column )
public static T Field<T>( this DataRow row, int columnIndex )
public static T Field<T>( this DataRow row, string columnName )
```

其中，参数 column 表示数据列（DataColumn），表示要返回数据的字段。参数 columnIndex 表示从 0 开始的列索引。columnName 表示要返回数据字段的名称。通常为了让代码更加通用，笔者建议尽量使用字段名称指定要返回的字段。

在示例代码 15-1 中，方法 BuildBookDataSet()在内存中创建一个名为 BookDataSet 的数据集合，它只包含一个名为 BookTable 的表。表 BookTable 中包含 4 个字段：int 类型的编号（ID）、string 类型的书名（Name）、string 类型的作者（Author）、double 类型的价格（Price）。

在方法 SelectBook()中，首先通过 BuildBookDataSet()创建数据集，然后通过 DataSet.Tables 属性获取名为 BookTable 的数据表。在查询 allBookQuery 和 bookPriceQuery 中，通过 DataTable.AsEnumerable()方法将 DataTable 转换成 IEnumerable<T>类型的数据集合，并进行查询。allBookQuery 查询集合中所有的书的信息，而 bookPriceQuery 只查询集合中所有书的书名和价格。

示例代码 15-1

```
static void SelectBook( )
{
    //获取数据集和数据表
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //查询 allBookQuery 表示查询 BookTable 中的所有书籍，演示 AsEnumerable()的使用
    var allBookQuery =
        from book in dt.AsEnumerable( )
        select book;
    //打印查询 allBookQuery 的结果
    System.Console.WriteLine("所有书的信息:");
    foreach (var item in allBookQuery)
    {
        //演示 Field<T>方法的使用
        System.Console.WriteLine("编号:{0:D3}  书名:{1, -15} 作者:{2, -8} 价
        格:{3}",
                                    item.Field<int>("ID"),
                                    item.Field<string>("Name"),
                                    item.Field<string>("Author"),
                                    item.Field<double>("Price"));
    }
    //查询 bookPriceQuery 表示查询 BookTable 中所有的书及其价格，演示 AsEnumerable
    ()和 Field<T>的使用
    var bookPriceQuery =
        from book in dt.AsEnumerable( )
        select new { Name = book.Field<string>("Name"), Price = book.Field
        <double>("Price") };
    //打印查询 bookPriceQuery 的结果
    System.Console.WriteLine("书名和价格:");
    foreach (var item in bookPriceQuery)
    {
        System.Console.WriteLine("书名:{0, -20} 价格:{1} ", item.Name, item.
        Price);
    }
    System.Console.WriteLine( );
}

static DataSet BuildBookDataSet( )
{
    //可选书名、作者、价格，用于创建数据
    string[] bookNames = { "重构--改善既有代码", "英语口语大全", "项目管理",
    "设计模式解析", "穷爸爸富爸爸" };
    string[] bookAuthors = { "张  三", "李  四", "王二麻", "黄  花", "杨  明"};
    double[] bookPrices = { 52.8, 23.5, 35.6, 48.8, 18.9 };
    //创建名为 BookDataSet 的 DataSet 对象
    DataSet ds = new DataSet("BookDataSet");
    //创建名为 BookTable 的 DataTable 对象，并添加到 BookDataSet 中
```

```
    DataTable dt = new DataTable("BookTable");
    ds.Tables.Add(dt);
    //创建 DataTable 的字段
    dt.Columns.AddRange(
        new DataColumn[]
        {
            new DataColumn("ID", typeof(int)),
            new DataColumn("Name", typeof(string)),
            new DataColumn("Author", typeof(string)),
            new DataColumn("Price", typeof(double)),
        });
    //填充数据
    for (int id = 0; id < bookNames.Length; id++)
    {
        //创建数据行的数据
        DataRow row = dt.NewRow( );
        row["ID"] = id;
        row["Name"] = bookNames[id];
        row["Author"] = bookAuthors[id];
        row["Price"] = bookPrices[id];
        //添加到数据表中
        dt.Rows.Add(row);
    }
    //返回 DataSet
    return ds;
}
```

示例代码 15-1 的输出如下，其中，查询 allBookQuery 的结果为表 BookTable 中所有书的信息，包括编号、书名、作者和价格。查询 bookPriceQuery 的结果只包括表 BookTable 中书的书名和价格。

```
所有书的信息:
编号:000  书名:重构--改善既有代码        作者:张  三      价格:52.8
编号:001  书名:英语口语大全              作者:李  四      价格:23.5
编号:002  书名:项目管理                  作者:王二麻      价格:35.6
编号:003  书名:设计模式解析              作者:黄  花      价格:48.8
编号:004  书名:穷爸爸富爸爸              作者:杨  明      价格:18.9
书名和价格:
书名:重构--改善既有代码          价格:52.8
书名:英语口语大全                价格:23.5
书名:项目管理                    价格:35.6
书名:设计模式解析                价格:48.8
书名:穷爸爸富爸爸                价格:18.9
```

技巧： 虽然 Field<T>()方法可以通过索引获取某个字段的值，但是由于索引在数据库设计改变后会发生变化，可能会导致异常，所以通常使用字段名获取某字段的值。

15.2.3　按指定条件过滤 DataTable 的记录

在 LINQ 查询中，where 子句用来对数据进行过滤，其同样也可以应用在 DataTable 上。一般地，wehere 子句通过 Filed<T>方法获取一个或多个字段的值，然后对这些值进行判断，从而达到对表中的记录进行过滤的功能。

如示例代码 15-2 所示，其中，查询 cheapBookQuery 使用 where 子句根据书的价格对书进行过滤，只需要价格小于 40 的书，然后再打印出查询结果。BuildBookDataSet()方法和示例代码 15-1 中一样。

示例代码 15-2

```
static void FilterBook( )
{
    //获取数据集和数据表
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //查询 cheapBookQuery 表示查询 BookTable 中的价格低于 40 的书
    var cheapBookQuery =
        from book in dt.AsEnumerable( )
        where book.Field<double>("Price") < 40.0
        select book;
    //打印查询 cheapBookQuery 的结果
    System.Console.WriteLine("价格低于 40 的书:");
    foreach (var item in cheapBookQuery)
    {
        //演示 Field<T>()方法的使用
        System.Console.WriteLine("编号:{0:D3}  书名:{1, -20} 作者:{2, -8} 价
        格:{3}",
                                    item.Field<int>("ID"),
                                    item.Field<string>("Name"),
                                    item.Field<string>("Author"),
                                    item.Field<double>("Price"));
    }
}
```

示例代码 15-2 的输出如下，其中，cheapBookQuery 输出所有价格低于 40 的书的所有信息，包括编号、书名、作者和价格。

```
价格低于 40 的书:
编号:001  书名:英语口语大全              作者:李  四      价格:23.5
编号:002  书名:项目管理                  作者:王二麻      价格:35.6
编号:004  书名:穷爸爸富爸爸              作者:杨  明      价格:18.9
```

15.2.4　按指定顺序排列 DataTable 的记录

在 LINQ 中，通过 orderby 子句对数据源中的数据进行排序，包括升序排序和降序排序两种，该子句同样可以应用在对 DataTable 的查询中。一般地，orderby 子句通过 Filed<T>方法获取一个或多个字段的值，然后对这些值排序，从而达到对表中的记录进行排序的功能。orderby 子句常和 where 子句一起使用，同时对数据表中的记录进行排序和过滤。

如示例代码 15-3 所示，BuildBookDataSet()方法和示例代码 15-1 一样，创建一个 BookDataSet 数据集。查询 priceBookQuery 查询表 BookTable 中所有的书，并用 orderby 子句将它们按价格从低到高的顺序排序。查询 cheapBookQuery 查询 BookTable 中所有价格低于 40 的书，同时用 orderby 子句将它们按价格从高到低排序。

示例代码 15-3

```
static void OrderBook( )
```

```
{
    //获取数据集和数据表
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //查询 priceBookQuery 表示查询 BookTable 中所有的书，且按照价格从低到高排序
    var priceBookQuery =
        from book in dt.AsEnumerable( )
        orderby book.Field<double>("Price")
        select book;
    //打印查询 priceBookQuery 的结果
    System.Console.WriteLine("所有书按价格从低到高排序:");
    foreach (var item in priceBookQuery)
    {
        //演示 Field<T>方法的使用
        System.Console.WriteLine("编号:{0:D3}  书名:{1, -20} 作者:{2, -8} 价
        格:{3}",
                                    item.Field<int>("ID"),
                                    item.Field<string>("Name"),
                                    item.Field<string>("Author"),
                                    item.Field<double>("Price"));
    }
    //查询 cheapBookQuery 表示查询 BookTable 中价格低于 40 的书，且按照价格从高到低
    排序
    var cheapBookQuery =
        from book in dt.AsEnumerable( )
        where book.Field<double>("Price") < 40.0
        orderby book.Field<double>("Price") descending
        select book;
    //打印查询 cheapBookQuery 的结果
    System.Console.WriteLine("价格低于 40 的书从高到低排序:");
    foreach (var item in cheapBookQuery)
    {
        //演示 Field<T>方法的使用
        System.Console.WriteLine("编号:{0:D3}  书名:{1, -20} 作者:{2, -8} 价
        格:{3}",
                                    item.Field<int>("ID"),
                                    item.Field<string>("Name"),
                                    item.Field<string>("Author"),
                                    item.Field<double>("Price"));
    }
}
```

示例代码 15-3 的输出如下，查询 priceBookQuery 将 BookTable 中所有的书按价格从低到高的顺序排序。查询 cheapBookQuery 将 BookTable 中所有价格低于 40 的书按价格从高到低排序。

```
所有书按价格从低到高排序:
编号:004   书名:穷爸爸富爸爸                作者:杨  明      价格:18.9
编号:001   书名:英语口语大全                作者:李  四      价格:23.5
编号:002   书名:项目管理                    作者:王二麻      价格:35.6
编号:003   书名:设计模式解析                作者:黄  花      价格:48.8
编号:000   书名:重构--改善既有代码          作者:张  三      价格:52.8
价格低于 40 的书从高到低排序:
编号:002   书名:项目管理                    作者:王二麻      价格:35.6
编号:001   书名:英语口语大全                作者:李  四      价格:23.5
编号:004   书名:穷爸爸富爸爸                作者:杨  明      价格:18.9
```

提示：除了 where 和 orderby 之外，在 DataTable 中还可以应用任何其他的 LINQ 运算，包括 Max()、Min()、Average()、Count()等，它们的语法与使用都与第 13 章类似，本章不再赘述。

15.2.5　用多个 from 子句查询多个 DataTable

通常，一个数据集（DataSet）包含多个数据表（DataTable），而且数据表之间具有一定的关联关系，从而表示一个关系型数据库。通过 LINQ to DataSet 同样可以轻松查询多个数据表中的数据，这需要使用多个 from 子句进行复合查询，同时通过 where 子句来进行多个表之间的关系判断。

本节的例子中，使用示例代码 15-4 中创建的数据集合，BuildDataSet()方法创建一个名为 StudentsDataSet 的数据表，包含两个数据表 StudentTable 和 ScoreTable。前者记录学生信息，包括姓名（Name）、性别（XingBie）、年龄（Age）、成绩号（ScoreID）。后者记录学生成绩，包括成绩号（ScoreID）、数学成绩（Math）、语文成绩（Chinese）、英语成绩（English）。其中，字段成绩号是两个表关联字段，同时该字段可以查询学生的成绩信息。

示例代码 15-4

```
static DataSet BuildDataSet( )
{
    //创建 Students 数据集
    DataSet ds = new DataSet("StudentsDataSet");
    //创建 Students 数据表，并添加到数据集
    //Students 数据表包含学生信息
    DataTable dtStu = new DataTable("StudentTable");
    ds.Tables.Add(dtStu);
    //添加学生信息记录的列信息
    dtStu.Columns.AddRange(new DataColumn[]{
        new DataColumn("Name", Type.GetType("System.String")),
        new DataColumn("XingBie", Type.GetType("System.String")),
        new DataColumn("Age", Type.GetType("System.Int32")),
        new DataColumn("ScoreID", Type.GetType("System.Int32")),
    });
    //添加学生信息的行信息
    dtStu.Rows.Add("张三", "男", 20, 1);
    dtStu.Rows.Add("李四", "男", 19, 2);
    dtStu.Rows.Add("王霞", "女", 21, 3);
    dtStu.Rows.Add("赵敏", "女", 22, 4);
    dtStu.Rows.Add("吴安", "男", 18, 5);
    //创建 Scores 数据表，并添加到数据集
    //Scores 数据表包含学生成绩记录
    DataTable dtScore = new DataTable("ScoreTable");
    ds.Tables.Add(dtScore);
    //添加成绩记录的列信息
    dtScore.Columns.AddRange(new DataColumn[]{
        new DataColumn("ScoreID", Type.GetType("System.Int32")),
        new DataColumn("Math", Type.GetType("System.Int32")),
        new DataColumn("Chinese", Type.GetType("System.Int32")),
        new DataColumn("English", Type.GetType("System.Int32")),
```

```
    });
    //添加学生成绩记录
    dtScore.Rows.Add(1, 80, 75, 78);
    dtScore.Rows.Add(3, 88, 80, 60);
    dtScore.Rows.Add(4, 75, 90, 80);
    dtScore.Rows.Add(5, 59, 80, 75);
    //返回数据集
    return ds;
}
```

查询多个数据表的数据通常通过多个 from 子句进行联合查询，每个 from 子句对应一个数据表，同时用 where 子句表示多个数据表之间的关系，一般单个 where 子句表示两个表之间的关系。在进行多表数据查询之前，要明确几个问题：

（1）要在哪些数据表中查询数据？from 子句该如何编写？

（2）查询结果包含哪些数据表的哪些字段？select 子句该如何编写？

（3）各数据表之间的关系如何进行关联？where 子句该如何编写？

（4）是否需要其他的操作，比如排序（orderby 子句）、分组（group 子句）等？

（5）该查询是使用简单的单个查询实现，还是通过多个查询组合实现？

如示例代码 15-5 所示，方法 QueryStuScores()中首先通过 BuildDataSet()方法获取数据集和要查询的数据表，其中 dtStu 表示学生信息数据表，dtScore 表示学生成绩数据表。查询 stuScores 用于查询数据集合中所有学生的成绩，如果学生没有成绩则不作为结果返回。

stuScores 中，第 1 个 from 子句从表 dtStu 中查询学生信息记录，并保存到临时变量 stu 中。第 2 个 from 子句从表 dtScore 中查询成绩记录，并保存到临时遍历 score 中。Where 子句则用于实现两个表之间的关联关系，即成绩号（ScoreID）相等。最后 select 子句则表示要将表 dtStu 的 Name 字段和 dtScore 的 Math、Chinese、English 字段作为查询结果。

示例代码 15-5

```
static void QueryStuScores( )
{
    //获取数据集和要进行查询的数据表
    DataSet ds = BuildDataSet( );
    DataTable dtStu = ds.Tables["StudentTable"];
    DataTable dtScore = ds.Tables["ScoreTable"];
    //查询 stuScores 查询所有学生的成绩
    var stuScores =
        from stu in dtStu.AsEnumerable( )
        from score in dtScore.AsEnumerable( )
        where stu.Field<int>("ScoreID") == score.Field<int>("ScoreID")
        select new
        {
            Name = stu.Field<string>("Name"),
            MathS = score.Field<int>("Math"),
            Chinese = score.Field<int>("Chinese"),
            English = score.Field<int>("English")
        };
    //打印查询 stuScores 的结果
    System.Console.WriteLine("所有学生成绩：");
    foreach (var item in stuScores)
    {
        System.Console.WriteLine("姓名:{0}, 数学:{1}, 语文:{2}, 英语:{3}",
            item.Name, item.MathS, item.Chinese, item.English);
    }
```

```
}
```

示例代码 15-5 的输出如下，从中可以看出学生“李四”没有成绩，所以不在查询 stuScores 的结果中。

```
所有学生成绩:
姓名:张三, 数学:80, 语文:75, 英语:78
姓名:王霞, 数学:88, 语文:80, 英语:60
姓名:赵敏, 数学:75, 语文:90, 英语:80
姓名:吴安, 数学:59, 语文:80, 英语:75
```

15.2.6　用 join 子句查询多个 DataTable

在 LINQ 中，查询多个数据源的数据，除了多个 from 子句外，还可以通过 join 子句实现多个数据表的联接，这和关系型数据（如 SQL Server）中的联接概念是相同的。本节同样使用示例代码 15-4 中创建的数据集 StudentsDataSet，演示如何通过 join 子句查询多个数据表中的数据。

如示例代码 15-6 所示，其代码和示例代码 15-5 相似，但查询 stuScore 不是使用两个 from 子句，而是使用 join 子句进行两个数据关联，这样就不需要 where 子句。另外，这里 stuScores 还对查询结果按照学生姓名升序排序。

示例代码 15-6

```
static void QueryStuScoresByJoin( )
{
    //获取数据集和要进行查询的数据表
    DataSet ds = BuildDataSet( );
    DataTable dtStu = ds.Tables["StudentTable"];
    DataTable dtScore = ds.Tables["ScoreTable"];
    //查询 stuScores 查询所有学生的成绩
    var stuScores =
        from stu in dtStu.AsEnumerable( )
        join score in dtScore.AsEnumerable( )
        on stu.Field<int>("ScoreID") equals score.Field<int>("ScoreID")
        orderby stu.Field<string>("Name")
        select new
        {
            Name = stu.Field<string>("Name"),
            MathS = score.Field<int>("Math"),
            Chinese = score.Field<int>("Chinese"),
            English = score.Field<int>("English")
        };
    //打印查询 stuScores 的结果
    System.Console.WriteLine("所有学生成绩: ");
    foreach (var item in stuScores)
    {
        System.Console.WriteLine("姓名:{0}, 数学:{1}, 语文:{2}, 英语:{3}",
            item.Name, item.MathS, item.Chinese, item.English);
    }
}
```

示例代码 15-6 的输出如下，和示例代码 15-5 的输出一样，只是根据学生姓名进行按照升序进行排序。

```
所有学生成绩:
姓名:王霞, 数学:88, 语文:80, 英语:60
姓名:吴安, 数学:59, 语文:80, 英语:75
姓名:张三, 数学:80, 语文:75, 英语:78
姓名:赵敏, 数学:75, 语文:90, 英语:80
```

技巧： join 子句一般可以用多个 from 子句结合多个 where 子句来替换，但是 join 子句效率更高，而且代码更简洁，所以在 LINQ to DataSet 查询中，应尽可能地使用 join 子句，而不是多个 from 子句。

15.2.7　用 DataRowComparer 比较数据

LINQ 定义了多种用于比较源元素的集合运算符，用来判断集合中的元素是否相等，包括 Distinct、Union、Intersect 和 Except。默认地，这些运算符通过对每个元素集合调用 GetHashCode()和 Equals()方法来比较源元素，这种比较方法对于普通的值类型数据完全可以正常工作。

对于 DataRow，通过 GetHashCode()执行引用比较，通常不能真正判断两行数据是否相等。因为对于数据表中的两行数据进行相等判断，通常需要确定各列中的元素值是否相等，而不是元素引用是否相等。为了解决这个问题，LINQ to DataSet 提供了 DataRowComparer 类，DataRowComparer 类不能直接实例化，而必须使用 Default 属性返回 DataRowComparer 的实例。然后调用 Equals(DataRow, DataRow)方法并作为输入参数传入要进行比较的两个 DataRow 对象。如果两个 DataRow 对象中排序的列值集合相等，则 Equals(DataRow, DataRow)方法返回 True，否则返回 False。

如示例代码 15-7 所示，方法 BuildDataTable()创建一个数据表，包括两个字段：编号（ID）和书名（Name）。方法 UseDataRowComparer()首先创建一个名为“书库 1”的书库，并增加一些书，再创建一个名为“书库 2”的书库，也增加一些书。最后，依次通过 Union()、Intersect()、Distinct()计算书库中所有书的并集、交集等。从中可以看出，只能通过 DataRowComparer.Default 静态属性来获取唯一的实例。

示例代码 15-7

```
static void Main(string[] args)
{
    UseDataRowComparer( );
}

public static DataTable BuildDataTable(string tableName)
{
    DataTable dt = new DataTable(tableName);
    dt.Columns.Add("ID", typeof(int));
    dt.Columns.Add("Name", typeof(string));
    return dt;
}

public static void UseDataRowComparer( )
{
    //创建第 1 个数据表用来表示一批书
    DataTable dt1 = BuildDataTable("书库 1");
```

```
    dt1.Rows.Add(new object[] { 1, "重构-改善现有代码结构" });
    dt1.Rows.Add(new object[] { 1, "重构-改善现有代码结构" });
    dt1.Rows.Add(new object[] { 2, "日常口语" });
    dt1.Rows.Add(new object[] { 3, "设计模式解析" });
    //创建第 2 个数据表用来表示另外一批书
    DataTable dt2 = BuildDataTable("书库 2");
    dt2.Rows.Add(new object[] { 1, "重构-改善现有代码结构" });
    dt2.Rows.Add(new object[] { 2, "突破英文词汇" });
    dt2.Rows.Add(new object[] { 3, "设计模式解析" });
    dt2.Rows.Add(new object[] { 4, "项目管理" });

    //查询并打印两个书库的并集
    var tableUnion = dt1.AsEnumerable( ).Union(dt2.AsEnumerable( ),
    DataRowComparer<DataRow>.Default);
    System.Console.WriteLine("书库 1 和书库 2 的并集");
    foreach (var item in tableUnion)
    {
        System.Console.WriteLine("    编号: {0}, 书名: {1}",
            item.Field<int>("ID"), item.Field<string>("Name"));
    }
    //查询并打印两个书库的交集
    var tableIntersect = dt1.AsEnumerable( ).Intersect(dt2.AsEnumerable( ),
    DataRowComparer<DataRow>.Default);
    System.Console.WriteLine("书库 1 和书库 2 的交集");
    foreach (var item in tableIntersect)
    {
        System.Console.WriteLine("    编号: {0}, 书名: {1}",
            item.Field<int>("ID"), item.Field<string>("Name"));
    }
    //查询并打印书库 1 中不重复的书
    var tableDistinct = dt1.AsEnumerable( ).Distinct(DataRowComparer
    <DataRow>.Default);
    System.Console.WriteLine("书库 1 不重复的书");
    foreach (var item in tableDistinct)
    {
        System.Console.WriteLine("    编号: {0}, 书名: {1}",
            item.Field<int>("ID"), item.Field<string>("Name"));
    }
}
```

示例代码 15-7 的输出如下，从中可以看出 DataRowComparer 类在比较 DataRow 时，的确是比较行中数据的所有字段的值，全部相等才认为是相等的。

```
书库 1 和书库 2 的并集
    编号: 1, 书名: 重构-改善现有代码结构
    编号: 2, 书名: 日常口语
    编号: 3, 书名: 设计模式解析
    编号: 2, 书名: 突破英文词汇
    编号: 4, 书名: 项目管理
书库 1 和书库 2 的交集
    编号: 1, 书名: 重构-改善现有代码结构
    编号: 3, 书名: 设计模式解析
书库 1 不重复的书
    编号: 1, 书名: 重构-改善现有代码结构
    编号: 2, 书名: 日常口语
    编号: 3, 书名: 设计模式解析
```

15.3　使用 LINQ to DataSet 修改数据

前面的示例代码中都是通过查询的方式使用 LINQ to DataSet，同样可以通过 DataRowExtensions 类的 SetField()方法修改 DataTable 中的数据。本节将介绍如何通过 LINQ to DataSet 修改数据。

15.3.1　修改 DataTable 中字段的值

在 LINQ to DataSet 中，DataRowExtensions 类提供泛型扩展方法 SetField()，用于设置数据表中指定列的数据，并且指定明确的数据类型。DataRowExtensions.SetField()方法具有 3 个重载版本，它们的定义如下：

```
public static void SetField<T>(
             this DataRow row,
             DataColumn column,
             T value)
public static void SetField<T>(
             this DataRow row,
             int columnIndex,
             T value)
public static void SetField<T>(
             this DataRow row,
             string columnName,
             T value)
```

其中，column 是表示要设置数据的列对象（DataColumn 类型），columnIndex 是从 0 开始的要设置数据的列索引，columnName 是要设置的数据列名称。

示例代码 15-8 演示 SetField()方法的使用，通过 BuildBookDataSet()方法创建书籍数据表 BookTable，第 1 个 foreach 语句遍历数据表的所有书。通过 Field<double>方法获取书的原价，然后通过 SetField<double>方法设置书的新价格。最后，打印出数据表中所有书籍的列表。

示例代码 15-8

```
static void AddBookPrice( )
{
   DataSet ds = BuildBookDataSet( );
   DataTable dt = ds.Tables["BookTable"];
   //通过 Field()获取价格，然后通过 SetField()设置价格
   foreach (var book in dt.AsEnumerable())
   {
      double orgPrice = book.Field<double>("Price");
      double newPrice = orgPrice + 2.5;
      book.SetField<double>("Price", newPrice);
   }

   //查询 allBookQuery 表示查询 BookTable 中的所有书籍，演示 AsEnumerable()的使用
   var allBookQuery =
      from book in dt.AsEnumerable( )
      select book;
```

```
    //打印查询 allBookQuery 的结果
    System.Console.WriteLine("所有书的信息:");
    foreach (var item in allBookQuery)
    {
        //演示 Field<T>方法的使用
        System.Console.WriteLine("编号:{0:D3}  书名:{1, -15} 作者:{2, -8} 价
        格:{3}",
                                  item.Field<int>("ID"),
                                  item.Field<string>("Name"),
                                  item.Field<string>("Author"),
                                  item.Field<double>("Price"));
    }
}
```

示例代码 15-8 的输出如下，从中可以看出，所有书的价格都增加了 2.5 元。

```
所有书的信息:
编号:000  书名:重构--改善既有代码     作者:张  三     价格:55.3
编号:001  书名:英语口语大全           作者:李  四     价格:26
编号:002  书名:项目管理               作者:王二麻     价格:38.1
编号:003  书名:设计模式解析           作者:黄  花     价格:51.3
编号:004  书名:穷爸爸富爸爸           作者:杨  明     价格:21.4
```

注意：SetField()方法是直接修改数据表中的记录，如果要保持原数据不变，那么在使用 SetField()之前，应该先备份源数据表，作者建议通过 CopyToDataTable()方法（15.3.2 节）对源数据表进行备份。

15.3.2　通过查询创建数据集备份

在 LINQ to DataSet 中，还可以通过 DataTableExtensions 类提供的扩展方法，CopyToDataTable()将从数据表中获取到的查询结果（类型为 IEnumerable<DataRow>）直接复制到一个新的数据表（DataTable）中，从而可以将查询结果绑定到界面控件（DataGridView 等），也可以使用一些 DataTable 特有的特性。

CopyToDataTable()包括 3 个重载版本，定义如下，其中第 1 个版本最简单，也最常用。注意，这里的所有类型 T 都是 DataRow 类型及其子类。

```
public static DataTable CopyToDataTable<T>(
              this IEnumerable<T> source) where T : DataRow
public static void CopyToDataTable<T>(
              this IEnumerable<T> source,
              DataTable table,
              LoadOption options) where T : DataRow
public static void CopyToDataTable<T>(
              this IEnumerable<T> source,
              DataTable table,
              LoadOption options,
              FillErrorEventHandler errorHandler) where T : DataRow
```

其中，table 表示目标数据表对象，用来保存数据。options 用于指定 DataTable 的加载属性。errorHandler 是一个函数委托，开发人员可以指定自定义的异常处理操作。CopyToDataTable()方法使用下面的过程通过查询创建 DataTable 复本。

（1）CopyToDataTable()方法复制源表中的 DataTable（实现 IQueryable<T>接口的 DataTable 对象）。IEnumerable 源通常来源于 LINQ to DataSet 表达式或方法查询。

（2）目标 DataTable 的架构从源表中第一个 DataRow 对象的列生成，复制表的名称是源表的名称加上单词 query。

（3）对于源表中的每一行，将行内容复制到新 DataRow 对象中，然后将该对象插入到目标 DataTale 中。

（4）复制完源表中所有 DataRow 对象后，返回复制的 DataTable。如果源序列不包含任何 DataRow 对象，则该方法将返回一个空 DataTable。

示例代码 15-9 演示 CopyToDataTable()方法的使用，其中，查询 query1 查询所有既有成绩，年龄又大于 20 岁的学生信息，此时 query1 类型为 IEnumerable<DataRow>。然后使用 query1 的 CopyToDataTable()方法创建一个 DataTable 副本 newDt，最后打印出 newDt 的数据。

示例代码 15-9

```
static void UseCopyToDTSimple( )
{
    //获取数据集和要进行查询的数据表
    DataSet ds = BuildDataSet( );
    DataTable dtStu = ds.Tables["Students"];
    DataTable dtScore = ds.Tables["Scores"];
    //查询 query1 年龄大于 20 且具有成绩的学生
    var query1 =
        from stu in dtStu.AsEnumerable( )
        from score in dtScore.AsEnumerable( )
        where stu.Field<int>("ScoreID") == score.Field<int>("ScoreID")
        where (int)stu["Age"] > 20
        select stu;
    //通过 CopyToDataTable()方法创建新的副本
    DataTable newDt = query1.CopyToDataTable<DataRow>( );
    //打印副本的信息
    System.Console.WriteLine("学生列表：");
    foreach (var item in newDt.AsEnumerable())
    {
        System.Console.WriteLine("姓名:{0}, 性别:{1},  年龄:{2}",
            item["Name"], item["XingBie"], item["Age"]);
    }
}
```

示例代码 15-9 的输出如下，可见 query1 的副本 newDt 所包含的数据和 query1 完全相同。

```
学生列表：
姓名:王霞, 性别:女,  年龄:21
姓名:赵敏, 性别:女,  年龄:22
```

15.4　使用 LINQ to DataSet 数据绑定

数据绑定是为用户展现数据的最佳方式，而 LINQ to DataSet 则是查询数据的方式，二者可以进行集成，即通过数据绑定技术将 LINQ to DataSet 查询的结果显示到用户界面。本

节就介绍该技术。

15.4.1　创建 DataView 数据源

数据绑定是在应用程序用户界面和业务逻辑之间建立连接的过程。如果绑定具有正确的设置，并且数据提供适当的通知，则在数据更改其值时，绑定到该数据的元素会自动反映更改。DataSet 是数据驻留在内存中的表示形式，不管包含的数据来自什么数据源，它都可以提供一致的关系编程模型。通过 DataView，可以使用不同的排序顺序公开表中的数据，并且可以按行状态或基于筛选器表达式来筛选数据。

根据前面的学习，可以知道 LINQ to DataSet 查询返回的是 DataRow 对象的集合，并不容易在数据绑定中使用它们。为了使得数据绑定更加容易，LINQ to DataSet 还提供了通过查询创建一个 DataView 的方式。它通过提供基于表达式的 LINQ 筛选和排序，扩展了 DataView 筛选和排序的功能，允许执行比基于字符串筛选和排序更复杂且功能更强大的筛选和排序操作。

在 LINQ to DataSe 中，DataView 并不是直接通过 new 运算符来创建，而是通过 AsDataView()扩展方法间接地创建 DataView 对象。AsDataView()方法包括 2 个重载版本，定义如下。由此可见，可以从 DataTable 对象或者 LINQ to DataSet 的查询结果创建 DataView 对象。

```
public static DataView AsDataView(
    this DataTable table
)
public static DataView AsDataView<T>(
    this EnumerableRowCollection<T> source
) where T : DataRow
```

在 AsDataView()的第 2 个重载版本中，需要 EnumerableRowCollection<DataRow>类型的数据源，该类型表示从 LINQ 查询返回的 DataRow 对象集合，可以从 LINQ 查询获取。

示例代码 15-10 演示 AsDataView()方法的使用。其中，BuildBookDataSet()只是用来创建一个包含许多书的数据源，在此不再赘述。在 CreateBookViewByTable()方法中，dvDt 是通过 DataTable 类的扩展方法 AsDataView()创建 DataView，该 DataView 中包含数据表 BookTable 中的所有记录。方法 CreateBookViewByQuery()则是通过 LINQ 查询 chipBookQuery 的扩展方法，创建 DataView 对象。

示例代码 15-10

```
static DataView CreateBookViewByTable( )
{
    //获取数据表对象
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //通过 DataTable 创建 DataView
    DataView dv = dt.AsDataView( );
    return dv;
}

static DataView CreateBookViewByQuery( )
{
```

```
    //获取数据表对象
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //建立 LINQ to DataSet 查询 chipBookQuery，只需要价格低于 40 的书
    var chipBookQuery =
        from book in dt.AsEnumerable( )
        where book.Field<double>("Price") < 40.0
        select book;
    //通过查询 chipBookQuery 创建 DataView 对象
    DataView dv = chipBookQuery.AsDataView( );
    return dv;
}

static DataSet BuildBookDataSet( )
{
    //可选书名、作者、价格，用于创建数据
    string[] bookNames = { "重构--改善既有代码", "英语口语大全", "项目管理",
    "设计模式解析", "穷爸爸富爸爸" };
    string[] bookAuthors = { "张  三", "李  四", "王二麻", "黄  花", "杨  明" };
    double[] bookPrices = { 52.8, 23.5, 35.6, 48.8, 18.9 };
    //创建名为 BookDataSet 的 DataSet 对象
    DataSet ds = new DataSet("BookDataSet");
    //创建名为 BookTable 的 DataTable 对象，并添加到 BookDataSet 中
    DataTable dt = new DataTable("BookTable");
    ds.Tables.Add(dt);
    //创建 DataTable 的字段
    dt.Columns.AddRange(
        new DataColumn[]
        {
            new DataColumn("ID", typeof(int)),
            new DataColumn("Name", typeof(string)),
            new DataColumn("Author", typeof(string)),
            new DataColumn("Price", typeof(double)),
        });
    //填充数据
    for (int id = 0; id < bookNames.Length; id++)
    {
        //创建数据行的数据
        DataRow row = dt.NewRow( );
        row["ID"] = id;
        row["Name"] = bookNames[id];
        row["Author"] = bookAuthors[id];
        row["Price"] = bookPrices[id];
        dt.Rows.Add(row);          //添加到数据表中
    }
    return ds;                    //返回 DataSet
}
```

注意：本例中只是创建 DataView 对象，并没有使用它，15.4.2 节将介绍如何将创建好的 DataView 对象绑定到用户界面上。

15.4.2　绑定 DataView 数据源到 DataGridView 控件

在前面 12 章介绍过数据绑定的内容，DataGridView 控件是用户界面中常用且功能强大的表格数据显示控件，本节将介绍如何将 DataView 数据源绑定到 DataGridView 控件。

将 DataView 绑定到 DataGridView 控件只需要两步即可，首先是根据需要创建 DataView 对象，然后在需要显示时设置 DataGridView 控件的 DataSource 属性为 DataView 对象即可。在这里，通过以下步骤创建实例 BindingDataView，它将演示 DataView 绑定到数据源的全部过程。

（1）创建一个 Windows 窗体程序，取名为 BindingDataView，并将它的主窗体改名为 FrmMain。

（2）在窗体上添加 1 个名为 dgvData 的 DataGridView 控件，并添加 1 个名为 btnReset 的按钮，并合理布局，如图 15-3 所示。

（3）响应窗体 FrmMain 的 Load 事件，在事件处理函数 FrmMain_Load()中创建要绑定的 DataView 对象，并将它绑定到 DataGridView 控件 dgvData 上。

（4）响应按钮 btnReset 添加 Click 事件，在事件处理函数 btnReset_Click()中同样创建并绑定 DataView 对象到 dgvData 上。

如示例代码 15-11 所示，CreateBookView()方法创建书库，并将所有书作为数据源创建 DataView 对象。在 FrmMain_Load()中通过代码“dgvData.DataSource = m_CurrentDataView;”将 DataView 对象 m_CurrentDataView 绑定到 DataGridView 控件 dgvData 中。

示例代码 15-11

```
private DataView CreateBookDataView( )
{
    //获取数据表对象
    DataSet ds = BuildBookDataSet( );
    DataTable dt = ds.Tables["BookTable"];
    //建立 LINQ to DataSet 查询 chipBookQuery，查询所有的书
    var chipBookQuery =
        from book in dt.AsEnumerable( )
        select book;
    //通过查询 chipBookQuery 创建 DataView 对象
    DataView dv = chipBookQuery.AsDataView( );
    return dv;
}
private DataView m CurrentDataView = null;

private void btnReload Click(object sender, EventArgs e)
{
    //获取 DataView 对象，并绑定对到 DataGridView 控件
    m CurrentDataView = CreateBookDataView( );
    dgvData.DataSource = m CurrentDataView;
}
private void FrmMain Load(object sender, EventArgs e)
{
    //获取 DataView 对象，并绑定对到 DataGridView 控件
    m CurrentDataView = CreateBookDataView( );
    dgvData.DataSource = m CurrentDataView;
}
```

实例 BindingDataView 的运行效果如图 15-3 所示。从图中可以看出，通过数据绑定，DataView 中的数据被很直观地显示在 DataGridView 控件上，而且可以进行添加、删除、修改、排序等操作。

15.4.3　筛选 DataView 数据的记录



图 15-3　BindingDataView 实例

实际的软件开发中，通常需要对 DataView 中的数据进行过滤操作。在 LINQ to DataSet 中有两种方法实现该功能，一种是通过具有 where 子句的 LINQ 查询结果创建 DataView，这样 DataView 中的数据本身就是已经过滤过的。另外一种是查询出所有记录，然后通过 DataView 的 RowFilter 属性指定具体的过滤条件。

DataView 类的 RowFilter 属性接收一个表示过滤条件的字符串，格式为指定列的名称后跟一个运算符和一个要筛选的值。运算符可以是等于（=）、大于（>）、小于（<）等。通过设置 RowFilter 属性的值，有两种不同的方式清除 DataView 上的过滤条件：

- 将 RowFilter 属性设置为 null。
- 将 RowFilter 属性设置为一个空字符串。

在这里，通过改善实例 BindingDataView，来演示如何使用 RowFilter 属性过滤 DataView 中的数据。首先通过以下步骤来修改 15.4.2 节中的实例 BindingDataView。

（1）添加 2 个按钮到窗体，分别取名为 btnFilter 和 btnUnFilter。

（2）响应 btnFilter 的 Click 事件，在事件处理函数 btnFilter_Click()中，设置 m_CurrentData View 的 RowFilter 属性为 Price<40。

（3）响应 btnUnFilter 的 Click 事件，在事件处理函数 btnUnFilter_Click()中，设置 m_CurrentDataView 的 RowFilter 属性为空字符串，如示例代码 15-12 所示。

示例代码 15-12

```
private void btnFilter_Click(object sender, EventArgs e)
{
    if (m_CurrentDataView != null)
    {
        //设置过滤条件为：只需要价格低于 40 的书
        m_CurrentDataView.RowFilter = "Price < 40";
    }
}
private void btnUnFilter_Click(object sender, EventArgs e)
{
    if (m_CurrentDataView != null)
    {
        //取消过滤条件
        m_CurrentDataView.RowFilter = "";
    }
}
```

修改后的实例 BindingDataView 运行之后，单击“过滤数据”按钮后得到如图 15-4 所示的效果。从图中可以看出，通过设置 DataView 类的 RowFilter()方法，的确可以过滤 DataView 中的数据。

🔔**技巧**：DataView.RowFilter 通常是在用户界面层次对数据显示进行简单过滤，对于复杂的数据过滤功能，笔者建议通过 LINQ 查询的 where 子句来实现。

15.4.4 排序 DataView 数据的记录



图 15-4 BindingDataView 实例过滤

通过单击 DataGridView 控件的列标题，可以对该列数据进行排序，但是它不能恢复到对所有行都不排序的状态，同时这也只能对单列数据进行排序。在实际开发中，也需要对 DataView 中的数据通过手动编码来排序。在 LINQ to DataSet 中有两种方法实现该功能，一种是通过具有 orderby 子句的 LINQ 查询创建 DataView，第二种是通过设置 DataView 的 Sort 属性设置要被排序的列。

DataView 类的 Sort 属性接收一个表示排序信息的字符串，它包含列名，后跟具体的排序方式，包括 ASC（升序）和 DESC（降序）。在默认情况下列按升序排序，多个列可用逗号隔开。同样地，清除 DataView 中的排序信息也有两种方式：

- ❑ 将 Sort 属性设置为 null。
- ❑ 将 Sort 属性设置为一个空字符串。

在这里，通过进一步改善实例 BindingDataView，来演示如何使用 Sort 属性排序 DataView 中的列。首先通过以下步骤修改 15.4.3 节中的实例 BindingDataView。

（1）添加 2 个按钮到窗体，分别取名为 btnFilter 和 btnUnFilter。

（2）响应 btnSort 的 Click 事件，在事件处理函数 btnSort_Click()中，设置 m_Current Data View 的 Sort 属性为“Price, Name”。

（3）响应 btnUnSort 的 Click 事件，在事件处理函数 btnUnSort_Click()中，设置 m_CurrentDataView 的 Sort 属性为空字符串，如示例代码 15-13 所示。

示例代码 15-13

```
private void btnSort Click(object sender, EventArgs e)
{
    if (m_CurrentDataView != null)
    {
        //首先按照价格，然后再按照书名排序
        m CurrentDataView.Sort = "Price, Name";
    }
}
private void btnUnSort Click(object sender, EventArgs e)
{
    if (m CurrentDataView != null)
    {
        //取消排序条件
        m CurrentDataView.Sort = "";
    }
}
```

修改后的实例 BindingDataView 运行之后，单击“排序数据”按钮后得到如图 15-5 所示的效果。从图中可以看出，首先按照书价从低到高排序，然后再按照书名从低到高排序。由此可见，通过设置 DataView 类的 Sort()方法的确可以对 DataView 中的数据进行排序，而且是任意列排序。



图 15-5　BindingDataView 实例过滤

技巧： 通过 DataView.Sort 属性通常是在用户界面层次对数据显示进行简单排序，对于复杂的数据排序功能，笔者建议通过 LINQ 查询的 orderby 子句来实现。

15.5　小　　结

LINQ to ADO.NET 是.NET 4.0 中 LINQ 的配套技术之一，它将 LINQ 和 ADO.NET 完美集成，从而实现对 DataSet 和数据库中数据的复杂查询。本章介绍了 LINQ to DataSet 的开发细节，通过本章的学习，读者应该掌握以下知识点：

- 什么是 LINQ to ADO.NET？
- 什么是 LINQ to DataSet？
- 如何用 LINQ to DataSet 查询 DataSet 中的数据？
- 如何用 LINQ to DataSet 修改 DataSet 中的数据？
- 如何用 DataRowComparer 比较 DataTable 中的多行数据是否相等？
- 如何通过 LINQ to DataSet 创建 DataView？
- 如何绑定 DataView 到 DataGridView 控件？
- 如何过滤 DataView 中的数据？
- 如何对 DataView 中的数据进行排序？

第 16 章　LINQ to SQL

LINQ to SQL 是 LINQ to ADO.NET 的另一个技术分支，它的重点是在对象关系模型的基础上直接对数据库中的数据进行查询。LINQ to SQL 可以根据对象关系模型将 LINQ 查询转换成对应的 SQL 语句从数据库查询，并将查询结果转换为内存中的对象，然后在内存中使用。本章将全面介绍 LINQ to SQL 的具体使用。

16.1　了解 LINQ to SQL

LINQ to SQL 以关系对象模型为基础，将关系型数据库中的表、字段等映射到内存中的对象模型，从而方便数据库与内存对象的交互。本节将介绍 LINQ to SQL 的基本知识。

16.1.1　了解 LINQ to SQL

LINQ to DataSet 本质上只是对内存中 DataSet 的查询，所以 LINQ 和 ADO.NET 只是一种简单的合作关系，并不能充分利用两个技术的优势。相反，LINQ to SQL 作为 LINQ to ADO.NET 的另外一个分支，它将关系数据库模型映射到开发人员所用的编程语言（如 C#）表示的对象模型中，这使得 LINQ 和 ADO.NET 集成得更加紧密。

LINQ to SQL 是 ADO.NET 系列技术的一部分，它基于由 ADO.NET 提供程序模型提供的服务。对象关系模型是 LINQ to SQL 的核心部分，它提供数据库模型和对象模型之间的映射关系。因此，开发人员可以将 LINQ to SQL 代码与现有的 ADO.NET 应用程序混合使用，如图 16-1 是 LINQ to SQL 的结构图。从图中可见，LINQ to SQL 同样是访问 DataSet 数据集合，但是它同时也和 DataAdapter、DataSet 集成，可以直接对数据库进行操作。LINQ to SQL 支持程序是该技术的一个核心之一，它负责将对象模型和关系数据库模型之间进行相互转换。

在对象关系模型（OR 模型）中，包含对象模型和关系模型两部分，其中关系模型指关系型数据库中数据的存储关系，包括数据集、数据表、字段。对象模型指数据库数据在内存中以 C#对象表示的形式，它和关系模型是一一对应的关系。关系模型中的数据表的定义（Table）在对象模型中用一个类来表示，表中每一列的定义用类的属性表示，数据表的数据用包含一个或多个类对象的集合表示。

LINQ to SQL 查询所用的语法与在 LINQ 中使用的语法相同，不同的是查询中引用的对象被映射到数据库中的元素。在应用程序执行期间，通过 LINQ to SQL 请求 LINQ 查询执行，LINQ to SQL 支持程序随后将 LINQ 查询转换成等效的 SQL 命令，并委托 ADO.NET 提供程序执行数据库操作。ADO.NET 提供程序将查询结果作为 DataReader 返回，然后 LINQ

to SQL 提供程序将 ADO.NET 结果转换成用户对象的 IQueryable 集合，并对 IQueryable 集合进行查询，返回查询结果。图 16-2 是 LINQ to SQL 查询的执行流程图。

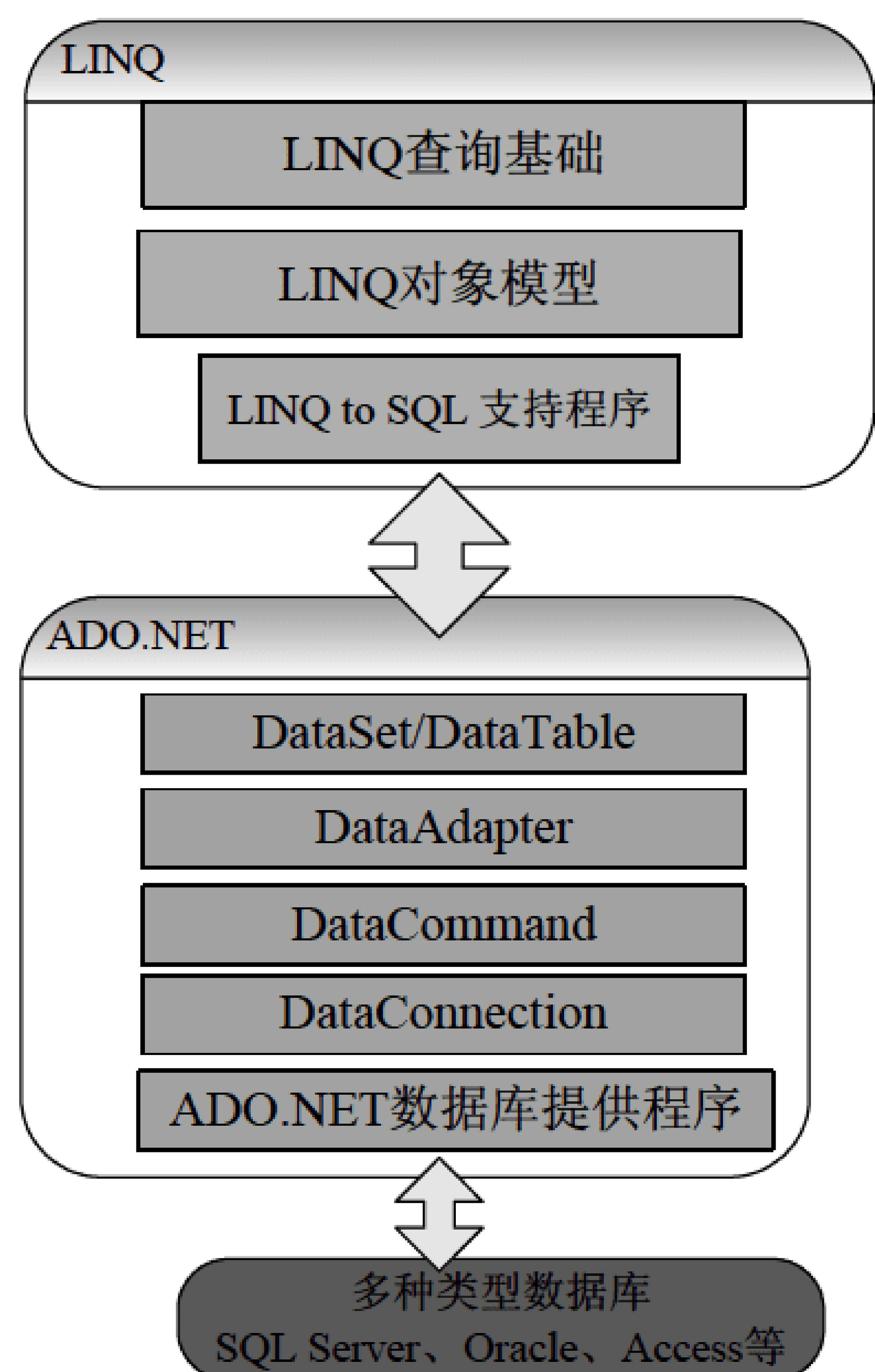


图 16-1　LINQ to SQL 和 ADO.NET

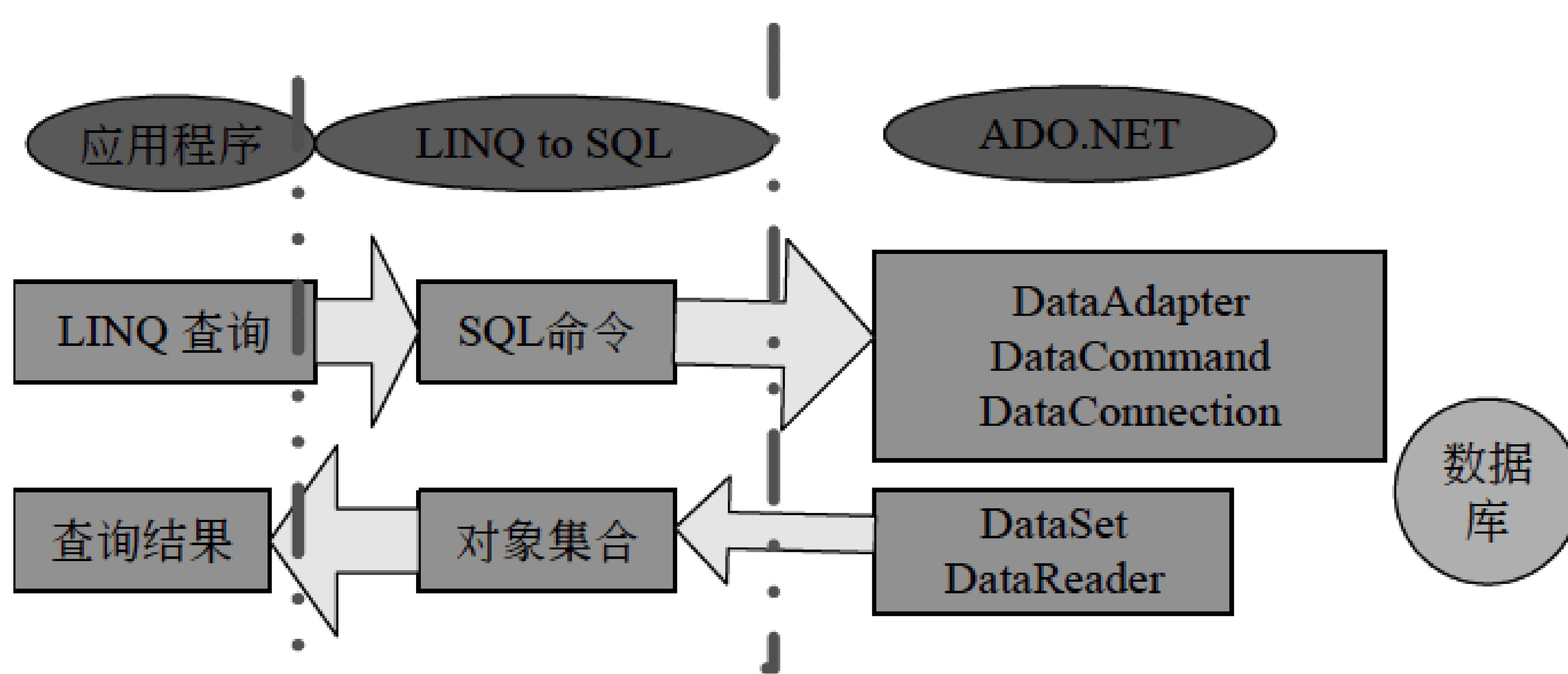


图 16-2　LINQ to SQL 查询执行流程

16.1.2　了解对象关系模型

对象关系（OR）模型，编程对象和数据库关系模型直接映射，比 LINQ to DataSet 更进一步。在 LINQ to SQL 中，将编程语言表示的对象模型映射到关系数据库的数据模型，开发人员按照对象模型执行对数据的操作，从而实现对数据库的操作。

在 LINQ to SQL 中，开发人员不用向数据库发出 SQL 命令（例如 SELECT、INSERT、UPDATE 等），而是在对象模型中直接更改值或执行方法。当开发人员通过对象模型执行操作时，LINQ to SQL 将请求转换成正确的 SQL 命令，然后将这些命令发送到数据库得到返回数据库服务器的操作结果后，再将这些结果传递到对象模型。

图 16-3 是 LINQ to SQL 对象模型原理图，从图中可以看出，LINQ to SQL 包括两个主要组件：对象模型（Object Model）和运行接口（Runtime）。前者为开发人员提供容易理解基于编程语言的对象访问模式和开发接口，后者为对象模型和数据库管理系统（DBMS，如 MS SQL Server 等）之间提供映射功能，将二者正确关联到一起。

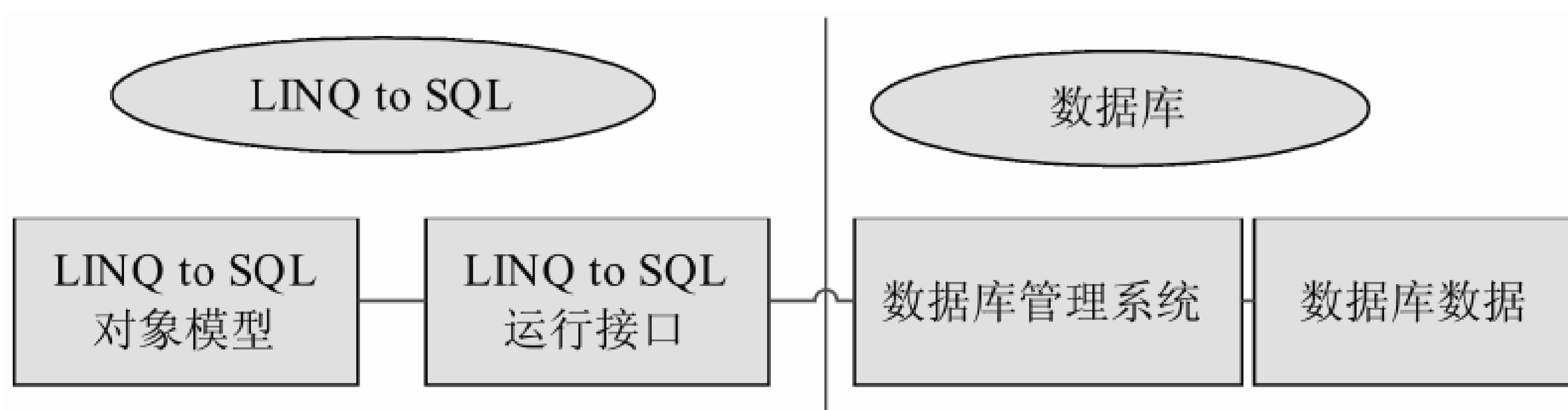


图 16-3　LINQ to SQL 模型

对象模型和关系型数据库中的元素之间有一种一对一的映射关系，开发人员才能通过对象模型访问数据库，表 16-1 给出了这种关系。其中，数据库中表（DataTable）被映射到 C#类（Class），数据库表的列（字段）被映射到 C#类的成员（属性），数据库表的外键关系（Relation）被映射到 C#类的关联，数据库表中的存储过程或函数被映射到 C#类的方法。当然，这些特定的 C#类及其成员和方法都会通过特定的属性（Attribute）修饰，将在 16.2 节详细介绍这些内容。

表 16-1　对象模型和关系数据库映射关系

对 象 模 型	关系型数据库模型
实体类	表
类成员	列
关联	外键关系
方法	存储过程或函数

通常，开发人员可以通过下面 3 种方法创建对象模型。

- ❑ 对象关系设计器：对象关系设计器（O/R 设计器）是 Visual Studio 2010 集成开发环境提供的一个工具，它为开发人员提供用于从现有数据库创建对象模型的丰富用户界面，最适合小型或中型数据库。
- ❑ SQLMetal 代码生成工具：SQLMetal 代码生成工具是一个实用的命令行工具，它的功能与 O/R 设计器十分相似，但是更加灵活。由于没有界面，可操作性要差很多，该工具适合对大型数据库进行建模。
- ❑ 代码编辑器：最后，开发人员还可以通过 Visual Studio 代码编辑器，或其他编辑器直接编写对象模型的代码。但是该方法编码量大，而且容易出错。

基于这 3 种方法各自的优缺点，通常在实际开发中，首先使用 O/R 设计器或 SQLMetal 代码生成工具创建初步的对象模型，然后，通过代码编辑器根据需要对对象模型进行改进或修改。

由于本书篇幅限制，本章在 16.2 节将对 O/R 设计器进行详细介绍，关于 SQLMetal 代码生成工具的更多知识，请读者查阅 MSDN 或相关书籍。

16.2　使用 O/R 设计器

O/R 设计器是 Visual Studio 2010 为开发人员提供的开发工具，它可以自动实现关系数据库模型到内存模型之间的转换，并且是可视化的设计界面，本节将介绍 OR 设计器的使用。

16.2.1　用 O/R 设计器创建 LINQ to SQL 类

Visual Studio 2010 为开发人员提供了一套自动化的对象关系设计器——O/R 设计器。O/R 设计器提供一个可视化设计界面，通过向导自动创建基于数据库中对象的 LINQ to SQL 实体类和关系，并且可以像编辑类图和数据库关系图那样编辑各个类、成员等。另外，O/R

设计器还生成一个强类型的 DataContext，用于在实体类与数据库之间发送和接收数据。

O/R 设计器还提供了相关功能，用于将存储过程和函数映射到 DataContext 方法以便返回数据和填充实体类。最后，O/R 设计器提供了对实体类之间的继承关系进行设计的能力。选中某个项目，通过右键菜单"添加"|"新建项"命令打开添加新项对话框，从中选择"LINQ to SQL 类"子项，如图 16-4 所示。然后，输入文件名称，这里为 UserLog，并单击"添加"按钮添加 LINQ to SQL 类。

从解决方案资源管理器视图中，可以看到项目下面新增的 LINQ to SQL 类——UserLog.dbml，它包括 4 个相关的文件，如图 16-5 所示。



图 16-4　添加 LINQ to SQL 类

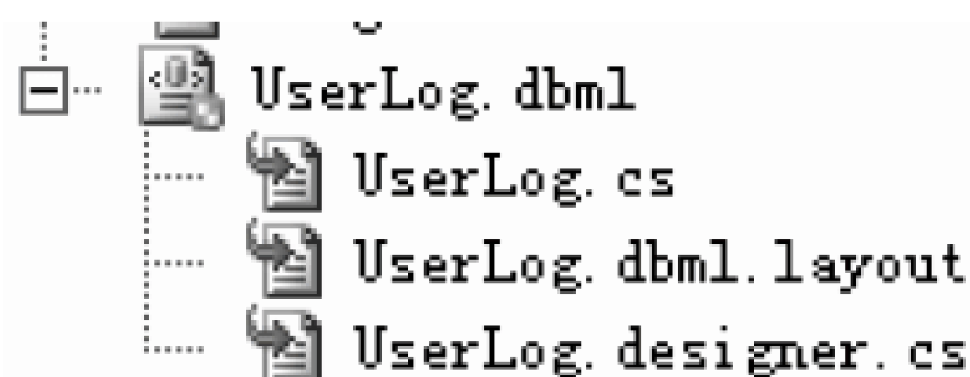


图 16-5　UserLog.dbml 文件结构

- *.dbml 文件：LINQ to SQL 类的基本文件，更多信息将在 16.2.2 节介绍。
- *.cs 文件：与当前 LINQ to SQL 类相关的后台代码文件，包含自定义代码，使用不多。
- *.dbml.layout：记录该 LINQ to SQL 类在 O/R 设计器中的布局信息，自动生成，建议不要修改。
- *.designer.cs：O/R 设计器自动生成的与 LINQ to SQL 类相关的类代码，包括 Adapter、DataSet 等，自动生成，建议不要修改。

双击 UserLog.dbml 可以打开 O/R 设计器视图，此时还没有任何数据。可以将"服务器资源管理器"中的数据库表直接拖到 O/R 设计器上，它会根据数据库中数据表的信息，自动添加对应的类、属性，包括类之间的关系。如图 16-6 所示，是本书中多处使用到的 UserLog 数据库中表 Logs 和 Users 在 O/R 设计器中的视图。

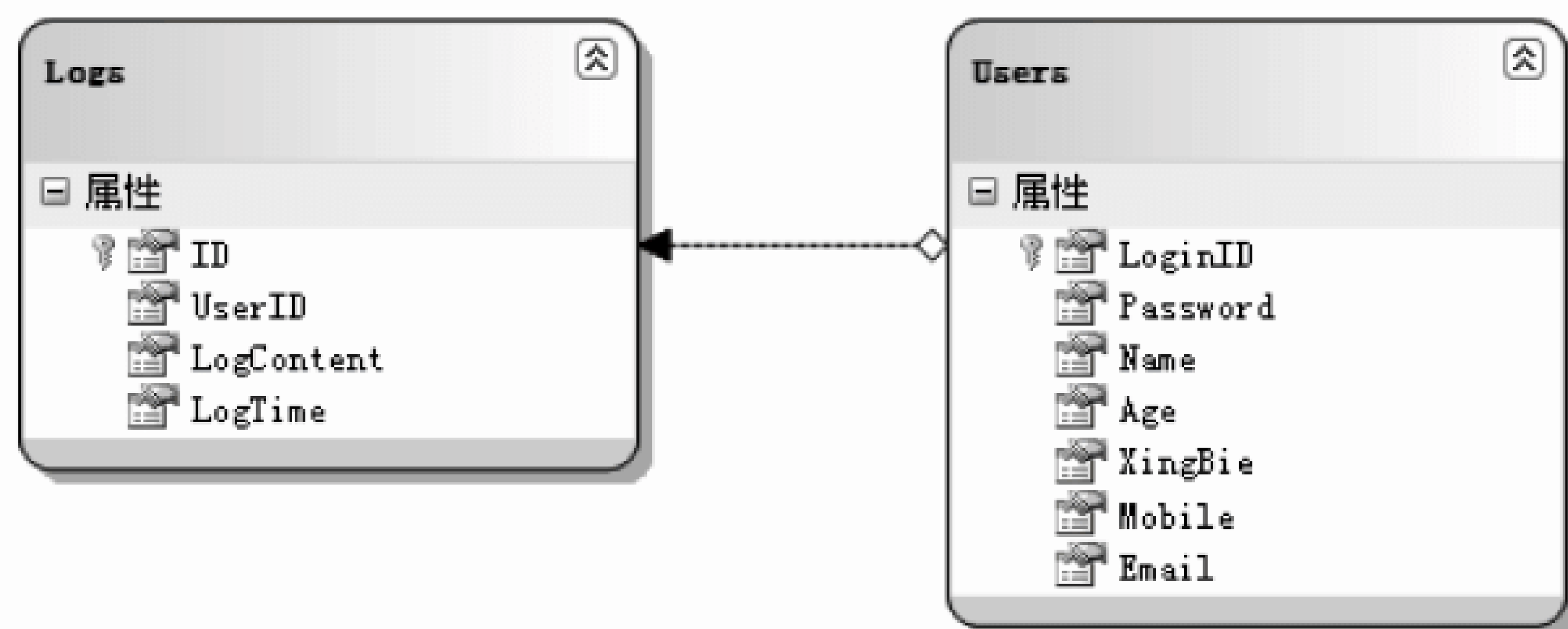


图 16-6　O/R 设计器视图

选中 O/R 设计器中任何一项，包括类、属性都可以从"属性管理器"视图中看到选中项的详细信息，如图 16-7 所示，是属性 UserID 的信息，从中可以看出一些属性和数据库

中参数是对应的，O/R 设计器可以自动从数据库中获取这些信息，并用这些信息来定义 LINQ to SQL 类。主要包括：

- Nullable：表示该属性对应的字段在数据库中是否可以为空。
- Server Data Type：表示该属性对应的字段在数据库中的数据类型。
- Source：表示该属性对应的字段在数据库中的列名。

Visual Studio 2010 为 O/R 设计器提供多个编辑工具，这在工具箱中可以看到，如图 16-8 所示。包括：类、关联、继承，这和类图设计类似，本质上它也是一个类图，所以可以添加新的类、为自动生成的类添加属性、方法等。



图 16-7　LINQ to SQL 属性信息

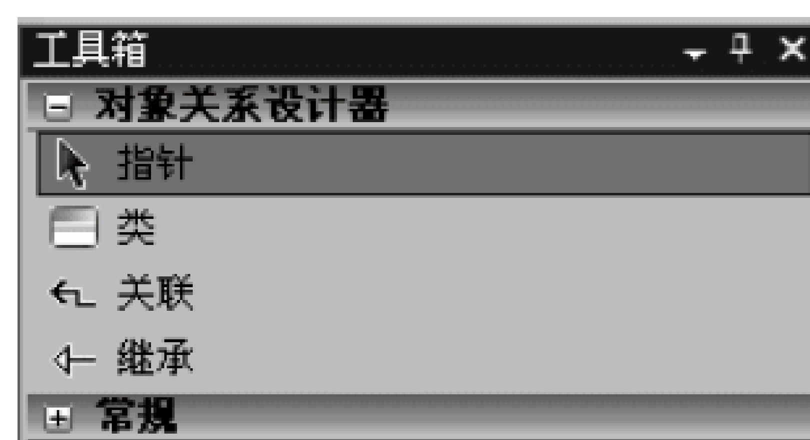


图 16-8　O/R 设计器工具箱

16.2.2　深入分析 DBML 文件

DBML 是 Database Mark Language 的缩写，中文为数据库标记语言。DBML 定义了 LINQ to SQL 类的元数据信息，O/R 设计器可以通过该文件自动创建对应的 C#代码，也就是前面提到的 CS 文件。

DBML 本身是一种 XML 语言的扩展，它记录关系数据库中成员（表、字段、关系）与内存对象（类、属性）之间的对应关系。如示例代码 16-1 所示，为 UserLog.dbml 的详细代码，从中可以看出它主要包括以下结点。

- Database 元素：表示关系型数据库与类的对应关系，包括类名（Name）和 DataContext 类（Class）。
- Connection 元素：表示数据库连接信息，包括连接字符串映射模式（Mode）、默认连接字符串（ConnectionString）。
- Table 元素：表示数据库表与类之间的对应关系，其中，Type 字结点定义了类信息。Name 属性和 Member 属性定义了数据库信息。
- Column 元素：定义了数据库表的字段与类的属性的对应关系，包括字段名（Name）、.NET 类型（Type）、数据库中数据类型（DbType）、是否为空等。

示例代码 16-1

```
<Database Name="UserLog" Class="UserLogDataContext" xmlns="http://schemas.
microsoft.com/linqtosql/dbml/2007">
```

```
  <Connection Mode="AppSettings" ConnectionString="Data Source=WWW-
818324B7DD9\YMYSQLSERVER;Initial Catalog=UserLog;Integrated Security=
True" SettingsObjectName="UseORDesiger.Properties.Settings" Settings
PropertyName="UserLogConnectionString"  Provider="System.Data.SqlClient"
/>
  <Table Name="dbo.Logs" Member="Logs">
    <Type Name="Logs">
      <Column Name="ID" Type="System.Int32" DbType="Int NOT NULL"
      IsPrimaryKey="True" CanBeNull="False" />
      <Column Name="UserID" Type="System.String" DbType="NVarChar(25) NOT
      NULL" CanBeNull="False" />
      <Column Name="LogContent" Type="System.String" DbType="NVarChar(1024)
      NOT NULL" CanBeNull="False" />
      <Column Name="LogTime" Type="System.DateTime" DbType="DateTime NOT
      NULL" CanBeNull="False" />
      <Association Name="Users_Logs" Member="Users" ThisKey="UserID" Type=
      "Users" IsForeignKey="True" />
    </Type>
  </Table>
  <Table Name="dbo.Users" Member="Users">
    <Type Name="Users">
      <Column Name="LoginID" Type="System.String" DbType="NVarChar(25) NOT
      NULL" IsPrimaryKey="True" CanBeNull="False" />
      <Column Name="Password" Type="System.String" DbType="NVarChar(25) NOT
      NULL" CanBeNull="False" />
      <Column Name="Name" Type="System.String" DbType="NVarChar(50)"
      CanBeNull="True" />
      <Column Name="Age" Type="System.Int32" DbType="Int NOT NULL"
      CanBeNull="False" />
      <Column Name="XingBie" Type="System.String" DbType="NVarChar(2)"
      CanBeNull="True" />
      <Column Name="Mobile" Type="System.String" DbType="NVarChar(20)"
      CanBeNull="True" />
      <Column Name="Email" Type="System.String" DbType="NVarChar(100)"
      CanBeNull="True" />
      <Association Name="Users_Logs" Member="Logs" OtherKey="UserID"
      Type="Logs" />
    </Type>
  </Table>
</Database>
```

一般情况下，DBML 文件都是通过 O/R 设计器自动生成，用户可以根据需要进行修改，但是通常不需要这样做。

16.3　LINQ to SQL 相关类

Visual Studio 2010 提供的 O/R 设计器，自动产生一系列 LINQ to SQL 正常工作所必需的类，从而组成对象模型，这些类实现数据库访问。本节将详细探讨这些 LINQ to SQL 相关类。

16.3.1　深入学习 DataContext 类

在 16.2 节介绍了如何通过 O/R 设计器创建 LINQ to SQL 类，也介绍了 DBML 文件的

结构，从 DBML 结构可以看到，这里并没有任何的类定义，那么这些类是定义在哪里？又是如何工作呢？本节将介绍自动生成的 LINQ to SQL 类的细节。

O/R 设计器自动生成的 LINQ to SQL 类代码在*.designer.cs 文件中，如 16.2.1 节的实例中，该文件是 UserLog.designer.cs。LINQ to SQL 类不仅仅是一个类，它是一组类的集合，数量取决于要操作的数据表的数量。如 UserLog.designer.cs 包括以下几个类。

- UserLogDataContext：DataContext 类是 LINQ to SQL 类与数据库进行通信的核心，所有的数据读取和写入都通过它来完成。
- Users：数据表 Users 对应的类，包括每个列的属性等。
- Logs：数据表 Logs 对应的类，同样包括每个列的属性等。

LINQ to SQL 通过 DataContext 类实现对象与数据库之间的数据交换，本示例中 DataContext 类的主要代码如示例代码 16-2 所示。UserLogDataContext 类从 DataContext 类继承而来，它根据指定的连接字符串自动创建特定的数据库连接，并进行数据通信。

示例代码 16-2

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="UserLog")]
public partial class UserLogDataContext : System.Data.Linq.DataContext
{
    private static System.Data.Linq.Mapping.MappingSource mappingSource =
    new AttributeMappingSource();
    #region Extensibility Method Definitions
    partial void OnCreated();
    partial void InsertLogs(Logs instance);
    partial void UpdateLogs(Logs instance);
    partial void DeleteLogs(Logs instance);
    partial void InsertUsers(Users instance);
    partial void UpdateUsers(Users instance);
    partial void DeleteUsers(Users instance);
    #endregion
    public UserLogDataContext() :
        base(global::UseORDesiger.Properties.Settings.Default.UserLog
        ConnectionString, mappingSource)
    {
        OnCreated();
    }
    //省略部分代码
    public System.Data.Linq.Table<Logs> Logs           //日志
    {
        get
        {
            return this.GetTable<Logs>();
        }
    }
    public System.Data.Linq.Table<Users> Users         //用户
    {
        get
        {
            return this.GetTable<Users>();
        }
    }
}
```

从示例代码 16-2 中可以看出，O/R 设计器自动生成的 DataContext 类，通常包含 4 类可供开发人员实现的部分（partial）方法。

- OnCreated()：该方法会在 DataContext 类启动之后自动调用，开发人员可以在这里对 DataContex 类进行初始化。
- InsertXX()：该方法是在插入某个表的数据之后自动调用，这里的 XX 表示数据库表名，如 InsertUsers()在往 Users 中插入记录后自动调用。
- UpdateXX()：该方法是在更新某个表的数据之后自动调用，这里的 XX 表示数据库表名，如 UpdateUsers()在更新 Users 中的记录后自动调用。
- DeleteXX()：该方法是在删除某个表的数据之后自动调用，这里的 XX 表示数据库表名，如 DeleteUsers()在删除 Users 中的记录后自动调用。

由此可见，开发人员通过扩展这 4 类方法，可以监视并控制着数据更新的全部过程，当然也可以不做任何事情。

技巧： 这里介绍的 OnCreated()等方法都是推荐用法，并不是绝对的。当完全手动开发一个 DataContext 类时，可以根据需要作出相应的调整。

16.3.2　深入学习数据表 Users 相关类

在 LINQ to SQL 相关的类中，与数据库表对应的类是另外一种核心类，每个数据表对应一个这样的类。每个数据表都表示为类，如示例代码 16-3 所示，为本示例中 Users 表对应的类。每个列在类中表示为一个字段和一个具有读写器的属性，数据类型取决于它在数据库中的类型。如字段 LoginID 在类 Users 中表示为字段_LoginID 和属性 LoginID。

通常，O/R 设计器自动生成的数据表类还包括几类可供开发人员扩展的方法，主要包括如下几个，通过这些方法，开发人员可以监视和控制数据的更新动作，并作出相应的处理。

- OnLoaded()：当数据表的数据加载完成后自动调用该方法。
- OnCreated()：当数据表对应的类被创建后自动调用该方法。
- OnXXChanging()：当数据表中某列对应的值更改前自动调用该方法，其中 XX 表示列名，如 OnLoginIDChanging()表示列 LoginID 即将更改。
- OnXXChanged()：当数据表中某列对应的值更改后自动调用该方法，其中 XX 表示列名，如 OnLoginIDChanged()表示列 LoginID 已经更改。

OnXXChanging()和 OnXXChanged 通常是在对应列的属性设置器中自动调用。如示例代码 16-3 中，在属性 LoginID 的设置器中，在设置之前首先调用 OnLoginIDChanging()方法，然后通过语句“this._LoginID=value;”设置列 LoginID 的新值，最后再调用 OnLoginIDChanged()方法。

示例代码 16-3

```
[Table(Name="dbo.Users")]
public partial class Users : INotifyPropertyChanging,
INotifyPropertyChanged
{
    private string _LoginID;
    private string _Password;
```

```
private string  Name;
private int  Age;
private string  XingBie;
private string  Mobile;
private string  Email;

#region Extensibility Method Definitions
partial void OnLoaded();
partial void OnValidate(System.Data.Linq.ChangeAction action);
partial void OnCreated();
partial void OnLoginIDChanging(string value);
partial void OnLoginIDChanged();
partial void OnPasswordChanging(string value);
partial void OnPasswordChanged();
partial void OnNameChanging(string value);
partial void OnNameChanged();
partial void OnAgeChanging(int value);
partial void OnAgeChanged();
partial void OnXingBieChanging(string value);
partial void OnXingBieChanged();
partial void OnMobileChanging(string value);
partial void OnMobileChanged();
partial void OnEmailChanging(string value);
partial void OnEmailChanged();
#endregion

public Users()
{
    //代码省略
    OnCreated();
}
 [Column(Storage="_LoginID", DbType="NVarChar(25) NOT NULL", CanBeNull=
False, IsPrimaryKey=True)]
public string LoginID
{
    get
    {
        return this. LoginID;
    }
    set
    {
        if ((this. LoginID != value))
        {
            this.OnLoginIDChanging(value);
            this.SendPropertyChanging();
            this. LoginID = value;
            this.SendPropertyChanged("LoginID");
            this.OnLoginIDChanged();
        }
    }
}
[Column(Storage="_Password", DbType="NVarChar(25) NOT NULL",
CanBeNull=False)]
public string Password
{
    //代码省略，类似于 LoginID
}
[Column(Storage=" Name", DbType="NVarChar(50)")]
public string Name
{
    //代码省略，类似于 LoginID
```

```
    }
    [Column(Storage=" Age", DbType="Int NOT NULL")]
    public int Age
    {
        //代码省略，类似于 LoginID
    }
    [Column(Storage="_XingBie", DbType="NVarChar(2)")]
    public string XingBie
    {
        //代码省略，类似于 LoginID
    }
    [Column(Storage=" Mobile", DbType="NVarChar(20)")]
    public string Mobile
    {
        //代码省略，类似于 LoginID
    }
    [Column(Storage="_Email", DbType="NVarChar(100)")]
    public string Email
    {
        //代码省略，类似于 LoginID
    }
    [Association(Name="Users Logs", Storage=" Logs", OtherKey="UserID")]
    public EntitySet<Logs> Logs
    {
        //代码省略，类似于 LoginID
    }
}
```

注意：同样地，这里提供的只是 O/R 设计器提供的推荐实现，如果开发人员手动编写这些代码，可以根据实际需要进行调整。

16.4　使用 LINQ to SQL 查询

LINQ to SQL 提供一种将关系型数据库映射到编程语言表示的对象模型，使得开发人员可以通过编程语言直接操作数据库，数据库访问变得更加快捷高效。本节将详细探讨 LINQ to SQL 查询的具体内容。

16.4.1　用 DataContext 加载数据

前面介绍过，DataContext 在 LINQ to SQL 中用于与数据库通信，并加载数据。通常不会直接使用 DataContext 类，而是从它派生出与某个数据库对应的 DataContext 类，如前面的 UserLogDataContext 类。一般需要在 DataContext 的构造函数中指定特定的数据库连接字符串，定义如下：

```
public DataContext(string fileOrServerOrConnection);
```

其中，参数 fileOrServerOrConnection 是指用于连接数据库的数据库连接字符串，或者需要访问的数据库文件。

DataContext 类创建好之后，就可以通过它的 GetTable()方法来获取某个数据表中的内容，该方法定义如下：

```
public ITable GetTable(Type type);
public Table<TEntity> GetTable<TEntity>() where TEntity : class;
```

其中，type 或 TEntity 表示与该数据表对应的内存中的类类型，如 16.3.2 节中的 Users、Logs 等。该方法将数据表中的记录读出来，并为每条记录创建一个内存对象，然后将记录中的数据保存在对象中。

如示例代码 16-4 所示，在按钮 btnLoad 的 Click 事件处理函数 btnLoad_Click()中，通过 UserLogDataContext 的构造函数创建一个 DataContext 对象。然后，通过它的 GetTable()方法获取数据表 Users 的记录。最后，将查询结果作为 DataGridView 控件的数据源，以便它可以正确地显示到界面。

示例代码 16-4

```
private void btnLoad_Click(object sender, EventArgs e)
{
    //用指定的数据库连接字符串，创建 DataContext 对象
    UserLogDataContext dataContext = new UserLogDataContext( @"Data
    Source=WWW-818324B7DD9\YMYSQLSERVER;Initial Catalog=UserLog;
     Integrated Security=True");
    //设置 DataGridView 控件的数据源，从而在界面显示数据
    this.dgvUsers.DataSource = dataContext.GetTable<Users>();
}
```

示例代码 16-4 的运行效果如图 16-9 所示，可见通过 DataContext 类访问数据库数据非常简单。值得注意的是，如果是通过 O/R 设计器自动生成的代码，通常它会自动从配置文件中获取连接字符串，也就是说可以通过默认构造函数创建 DataContext 对象。如示例代码 16-4 可以写成如示例代码 16-5 所示的代码。

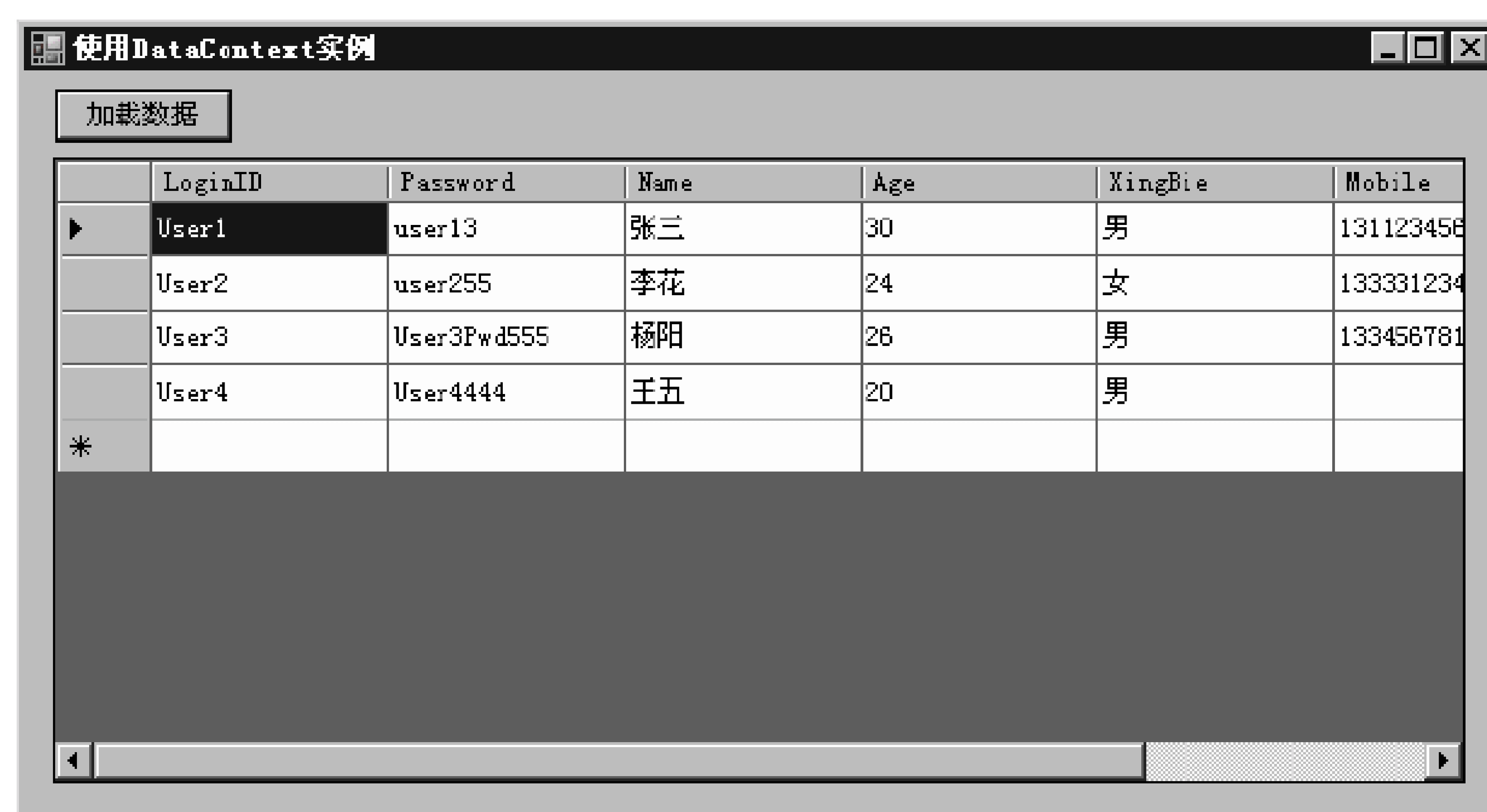


图 16-9　DataContext 实例运行效果图

示例代码 16-5

```
private void btnLoad_Click(object sender, EventArgs e)
{
    //从配置文件中获取数据库连接字符串，创建 DataContext 对象
    UserLogDataContext dataContext = new UserLogDataContext();
    //设置 DataGridView 控件的数据源，从而在界面上显示数据
    this.dgvUsers.DataSource = dataContext.GetTable<Users>();
}
```

16.4.2　查询数据库单表记录

顾名思义，可以通过 LINQ 对 LINQ to SQL 类进行查询，从而实现高效快速的数据库记录查询操作，并编写功能强大而普通 SQL 查询无法实现的数据查询功能。要用 LINQ 通过 LINQ to SQL 类查询数据库数据，通常包含以下 4 个步骤：

（1）在已有数据库的基础上，为项目添加 LINQ to SQL 对象模型。

（2）根据对象模型，获取对应的 DataContext 类。

（3）通过 DataContext 对象获取对应的数据源，根据对象模型不同，数据源保存的属性也不同。

（4）编写 LINQ 查询，对数据源进行查询。

O/R 设计器封装的 DataContext 类本身提供的数据表记录集合，可以被 LINQ 直接查询，例如 16.4.1 节中实例中的 UserLogDataContext 类提供的 Users 就可以作为数据源进行查询。如示例代码 16-6 所示，通过简单的 LINQ 代码查询数据表 Users（dataContext.Users）中年龄大于 20 的用户。

示例代码 16-6

```
private void btnAge20 Click(object sender, EventArgs e)
{
    //从配置文件中获取数据库连接字符串，创建 DataContext 对象
    UserLogDataContext dataContext = new UserLogDataContext( );
    //查询只需要年龄大于 20 岁的用户
    var oldusrs =
        from user in dataContext.Users
        where user.Age > 20
        select user;
    //将查询结果作为控件 dgvUsers 的数据源
    this.dgvUsers.DataSource = oldusrs;
}
```

示例代码 16-6 的输出如图 16-10 所示，与图 16-9 相比，它只列出了年龄大于 20 的用户。由此可见，可以将查询结果作为 DataGridView 控件的数据源直接绑定，同时也可以通过 LINQ 查询执行相当复杂的查询。

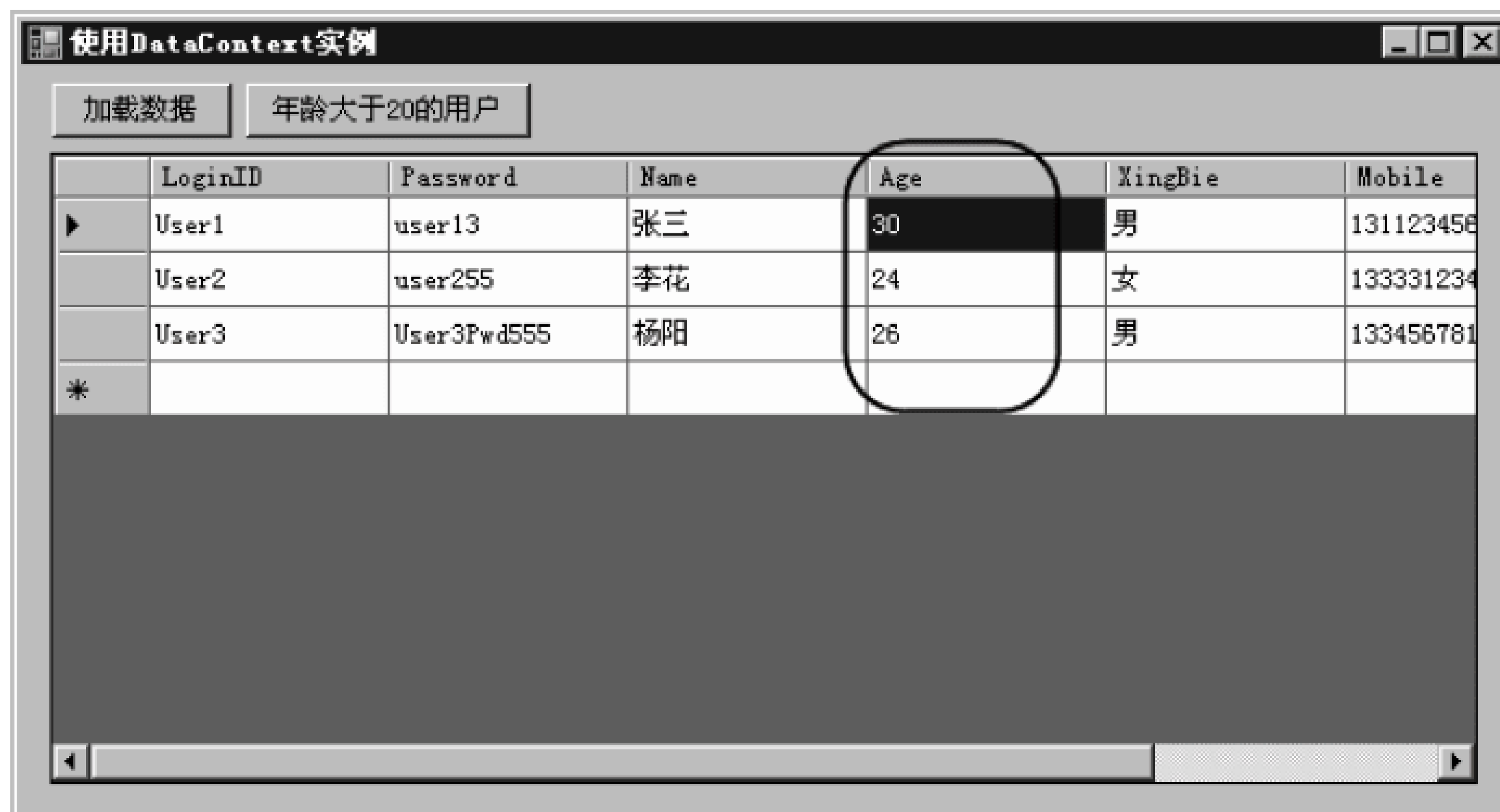


图 16-10　年龄大于 20 的用户

16.4.3　查询数据库多表记录

通过 LINQ 可以对 LINQ to SQL 类进行任何 LINQ 支持的查询，包括过滤、排序、联接等，可以查询单个数据表，也可以查询多个数据表。只需要将 DataContext 中的多个数据表作为 LINQ 查询的数据源即可。

如示例代码 16-7 所示，查询 oldUserLog 就是一个典型的多表查询实例，它通过 2 个 from 子句对表 Users 和 Logs 执行联接查询，查询结果年龄大于 20 的用户的所有日志，而且通过匿名类型过滤需要显示的信息。

示例代码 16-7

```
private void btnUserLog_Click(object sender, EventArgs e)
{
    //从配置文件中读取数据库连接字符串，创建 DataContext 对象
    UserLogDataContext dataContext = new UserLogDataContext( );
    //查询年龄大于 20 的用户的所有记录
    var oldUserLog =
        from user in dataContext.Users
        from log in dataContext.Logs
        where user.Age > 20
        select new { Name = user.Name, Age = user.Age,
                     Time = log.LogTime, Content = log.LogContent };
    //设置 DataGridView 控件的数据源，从而在界面上显示数据
    this.dgvUsers.DataSource = oldUserLog;
}
```

示例代码 16-7 的输出如图 16-11 所示。

使用DataContext实例

加载数据　年龄大于20的用户　用户日志

Name	Age	Time	Content
张三	30	2009-2-3 22:26	User1的留言---1
张三	30	2001-1-2 15:22	User1的留言---2
张三	30	2002-1-3 14:20	User2的留言---1
李花	24	2009-2-3 22:26	User1的留言---1
李花	24	2001-1-2 15:22	User1的留言---2
李花	24	2002-1-3 14:20	User2的留言---1
杨阳	26	2009-2-3 22:26	User1的留言---1
杨阳	26	2001-1-2 15:22	User1的留言---2
杨阳	26	2002-1-3 14:20	User2的留言---1

图 16-11　年龄大于 20 的用户的日志

注意： LINQ 对 DataContext 提供的数据源进行查询都是直接从转换成 SQL 命令执行，所以效率非常高，而且也不会查询任何多余的数据。这和 LINQ to DataSet 完全不同。

16.4.4　修改数据库的记录

DataContext 类作为对象模型和数据库模型之间的桥梁，在提供读取数据库接口的同时

还提供了数据库修改接口。通过 DataContext 类修改数据通常包含以下 4 个步骤：

（1）创建对象关系模型，创建和数据库对应的 DataContext 类型。

（2）通过 DataContext 类提供的数据表对象获取数据记录，

（3）通过设置类的属性修改记录的值。

（4）通过 DataContext 类的 SubmitChanges()方法将修改后的数据提交到数据库，SubmitChanges()方法包括两个重载版本，定义如下：

```
public void SubmitChanges();
public virtual void SubmitChanges(ConflictMode failureMode);
```

其中，参数 failureMode 是枚举类型 ConflictMode，表示提交过程中发生并发错误时要采取的操作，包括两个可选参数，FailOnFirstConflict 为默认值。

- FailOnFirstConflict：当检测到第一个并发冲突错误时，应立即停止对更新数据库的尝试。
- ContinueOnConflict：尝试对数据库的所有更新，并且应在该过程结束时累积和返回并发冲突。

如示例代码 16-8 所示，首先创建熟悉的 UserLogDataContext 对象，然后通过 foreach 语句遍历表 Users 中的所有记录，并将所有用户的年龄加 1。最后，通过 SubmitChanges()方法将更改提交到数据库，并将数据表 Users 作为 dgvUsers 的数据源。

示例代码 16-8

```
private void btnAgeInc Click(object sender, EventArgs e)
{
    //从配置文件中读取数据库连接字符串，创建 DataContext 对象
    UserLogDataContext dataContext = new UserLogDataContext( );
    //遍历所有的用户，并将他们的年龄加 1
    foreach (var user in dataContext.Users)
    {
        user.Age = user.Age + 1;
    }
    //提交更改
    dataContext.SubmitChanges( );
    //设置 DataGridView 控件的数据源，从而在界面显示数据
    this.dgvUsers.DataSource = dataContext.Users;
}
```

示例代码 16-8 的执行之后，得到如图 16-12 所示的效果图，与图 16-9 比较可以看出，所有用户的年龄都增加了 1 岁，直接到数据表查看也可以得到同样结论。

图 16-12　年龄加 1 后的用户

注意：在通过 DataContext.SubmitChanges()提交更新时，如果新数据有冲突，该方法会根据指定模式抛出异常。开发人员可以通过捕获这些异常来判断提交是否成功。

16.5　小　　结

LINQ to SQL 作为 LINQ to ADO.NET 的另外一个核心组件，它利用对象关系模型将数据库中的表、字段等映射到内存中的类、属性等。LINQ to SQL 通过将对对象模型的 LINQ 查询转换成对应的对数据库的 SQL 命令，并在服务器端执行 SQL 命令，从而达到 LINQ 查询数据库的功能。

本章深入介绍了 LINQ to SQL 技术，以及如何通过它来实现数据库的读取和修改等。通过本章的学习，读者应该掌握以下知识点：

- 什么是 LINQ to SQL？它有什么优点？
- 什么是对象关系模型？
- 如何使用 O/R 设计器创建数据关系模型？
- 什么是 DataContext 类？它有什么功能？
- 如何通过 DataContext 类与数据库通信？
- 如何通过 LINQ to SQL 查询单表数据？
- 如何通过 LINQ to SQL 查询多表数据？
- 如何通过 LINQ to SQL 修改并提交数据？

第 17 章　LINQ to XML

XML 已经被广泛使用在包括网络、数据存储、参数配置、数据库等许多领域，对 XML 数据的查询也越来越成为一个重要的话题。LINQ to XML 提供使用 LINQ 在内存中操作 XML 数据的编程接口，相当于更新的和重新设计的文档对象模型 XML 编程接口。本章将介绍如何通过 LINQ to XML 查询和编辑 XML 数据。

17.1　了解 XML

由于 XML 的广泛使用，.NET 提供了对 XML 的全面支持。在介绍 LINQ to XML 之前，本节先花少量篇幅介绍 XML 的基本概念，以及实际开发中常用的 DOM 方式操作 XML 数据会用到的一些基本数据类型。

17.1.1　了解 XML 文件

XML 是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。XML 也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

HTML 或格式化的程序语言，只是定义一套固定的标记，用来描述一定数目的元素，如果标记语言中没有所需的标记，用户也就没有办法了。XML 解决了这个缺陷，它是一种元标记语言，用户可以定义自己需要的标记。这些标记必须根据某些通用的规则来创建，但是在标记的意义上，也具有相当的灵活性。

假如用户正在处理与用户日志有关的事情，需要描述用户的编号、姓名、年龄、性别、电话、E-mail，这就必须创建用于用户的标记。新创建的标记可在文档类型定义（Document Type Definition，DTD）中加以描述。现在，只需把 DTD 看作是一本词汇表和某类文档的句法。XML 标记描述的是文档的结构和意义，它不描述页面元素的格式化，可用样式单为文档增加格式化信息。文档本身只说明文档包括什么标记，而不是说明文档看起来是什么样的。

XML 规范定义了 XML 文件的编写格式，XML 文件以树状结构描述一个文件中的数据，每个数据都有一个自定义的标识，还可以添加属性和文本元素。例如前面人事管理用 XML 格式编写，并命名为 UserList.xml，内容如示例代码 17-1 所示。其中，<?...?>表示 XML 文件整体的一些说明。标识分成两级：UserList 和 User，UserList 表示一组用户，具有一个属性 Count，表示包含几个用户。User 表示用户，具有属性 ID、Name、Age、XingBie、

Telphone、E-mail 元素。由于篇幅限制，这里不对 XML 的具体规范和格式做详细讲解，需要可以参看相关书籍，或到 XML 官方网站阅读。

示例代码 17-1

```
<?xml version="1.0" encoding="utf-8" ?>
<UserList Count="3">
  <User ID="1">
    <Name>张三</Name>
    <Age>18</Age>
    <XingBie>男</XingBie>
    <Telphone>13112345678</Telphone>
    <Email>zhangsan@126.com</Email>
  </User>
  <User ID="2">
    <Name>李四</Name>
    <Age>25</Age>
    <XingBie>男</XingBie>
    <Telphone>13112348888</Telphone>
    <Email>lisi@126.com</Email>
  </User>
  <User ID="3">
    <Name>黄花</Name>
    <Age>22</Age>
    <XingBie>女</XingBie>
    <Telphone>13112346666</Telphone>
    <Email>huanghua@126.com</Email>
  </User>
</UserList>
```

Visual Studio 2010 提供了 XML 文件编辑工具，通过它可以方便地创建和修改 XML 文件，可以创建 XML 文件的数据结构（本书不做详细介绍）。

XML 文件是符合 XML 规范的文本文件，它以“*.xml”作为文件扩展名。XML 既可以作为 HTML 文件的补充，用于网页开发和网络数据传输，也可作为本地文件用来存储数据，在数据量不大的情况下，它远比数据库要简单和快捷得多。本章将 XML 作为本地数据存储文件，介绍如何在 C# 3.0 中读写 XML 文件中的数据。

17.1.2　了解 System.Xml 命名空间

在.NET 类库中对 XML 文件的访问提供了强大的支持，与 XML 访问相关的类都被封装在 System.Xml 命名空间下，它根据功能被细分成 4 个子命名空间：System.XML.Schema、System.XML.Serialization、System.XML.Xpath 和 System.XML.Xsl。

由于.NET 对 XML 规范进行完整的支持，内容十分丰富和繁多，本章不可能覆盖所有相关内容，重点讲解 XML 文件作为数据存储文件时如何进行数据存取。通过.NET 类库提供的以下几个类可完成 XML 文件中数据的存取。

- XmlElement：表示 XML 文档中的一个元素，如前面的<Age>24</Age>。
- XmlEntity：表示 XML 文档中的一个实体声明，格式为<!ENTITY…>。
- XmlAttribute：表示 XML 文档中某元素的属性，如前面 Company 元素的 Name

属性。

- XmlText：表示 XML 文档中的文本，如“<Gender>男</Gender>”中的文本“男”。
- XmlDeclaration：表示 XML 文档的声明结点，格式为<?xml ver="1.0"……?>。
- XmlComment：表示 XML 文档中的一段注释，格式为<--注释文本-->
- XmlDocment：表示 XML 文档，在内存中以树状形式保存 XML 文档中的数据。
- XmlNode：表示 XML 文档中的一个结点。
- XmlNodeType：枚举类型，表示 XmlNode 的具体类型。比如元素开始、元素结束、属性、文本、空白等。
- XmlReader：表示一个读取器，它以一种快速、非缓存和只进的方式读取包含 XML 数据的流或文件。
- XmlWriter：表示一个编写器，它以一种快速、非缓存和只进的方式生成包含 XML 数据的流或文件。
- ReadState：表示读取器的读取状态。
- WriteState：表示编写器的写入状态。

System.Xml 命名空间及其子命名空间下提供的类和操作远不止上面几个，但本章只介绍其中几个常用的类及其成员，用它们访问 XML 文件中的数据。一些 XML 文件处理更高级的技术，可以从 MSDN 或相关书籍上获得更多详细的介绍。

注意：*在使用这些 XML 文件操作类之前，要先引用命名空间 System.Xml 以及对应的子命名空间，代码如下：*

```
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;
using System.Xml.XPath;
using System.Xml.Xsl;
```

17.2　使用 DOM 操作 XML 数据

文档对象模型（DOM）是常见的操作 XML 数据技术，它将 XML 数据表示为内存中的树型结构表示对象集合，然后会对这些对象进行查询和编辑等操作，并且可以还支持保存到文件。本节介绍 DOM 的一些使用细节。

17.2.1　用 XmlReader 读取 XML 数据

在一些软件开发中，只是需要简单的读取和写入 XML 数据，这时就可以使用 DOM 提供的 XmlReader 和 XmlWriter 两个类。XmlReader 类表示一个读取器，它以一种快速、非缓存和只进的方式读取包含 XML 数据的流或文件。恰好相反，XmlWriter 类提供一种快速、非缓存和只进的方式来生成包含 XML 数据的流或文件。

XmlReader 提供了大量用于读取 XML 数据的属性和方法，本节通过简单的示例来演示如何使用这些属性和方法读取 XML 数据，首先需要了解 XmlReader 类的几个常用属性

和方法。

```
public abstract bool EOF { get; }
```

XmlReader.EOF 为只读属性，表示 XmlReader 的当前读取位置是否在数据流结尾，如果是 XML 文件，则表示文件结尾。

```
public abstract XmlNodeType NodeType { get; }
public virtual string Name { get; }
```

XmlReader.NodeType 和 XmlReader.Name 分别表示 XmlReader 所读取的当前结点的结点类型和限定名，根据结点类型不同，结点具有不同的限定名，如表 17-1 所示。

表 17-1　NodeType与Name对应表

结点类型（NodeType）	限定名（Name）
Attribute	属性的名称，如 UserList.xml 中 UserList 元素的属性 Count
DocumentType	文档类型的名称
Element	元素的标记名，如 UserList.xml 中的 UserList
EntityRefrence	引用实体的名称
XmlDeclaration	固定的字符串 xml

Create()静态方法用于在具有 XML 数据的文件或数据流上，创建并返回一个 XmlReader 读取器，它常用的重载包括以下 3 个：

```
public static XmlReader Create(string inputUri);
public static XmlReader Create(Stream input);
public static XmlReader Create(TextReader input);
```

其中，inputUri 表示包含 XML 数据的 URL 地址，可以是一个文件名，也可以是网络上的 URL 地址。input 表示包含 XML 数据的数据流，可以是任何包含 XML 数据的流，也可以是 TextReader 流。

```
public abstract bool Read();
```

XmlReader.Read()从数据流中读取下一个 XML 结点，如果读取成功返回 True，否则返回 False。通常通过该方法的返回值来判断是否达到数据流结尾。

XmlReader.MoveToAttribute()将读取器移到当前元素的下一个指定的属性，所以，在使用该方法之前要先通过 Read()等方法读取到一个具有属性的结点。常用的 2 个定义如下：

```
public virtual void MoveToAttribute(int index);
public abstract bool MoveToAttribute(string name);
```

其中，index 表示目标属性在当前结点属性列表中从 0 开始的索引号。name 表示目标属性的限定名，注意，同一个 XML 元素的属性必须具有不同的名称。MoveToFirstAttribute()和 MoveToLastAttribute()方法也常用来遍历元素的属性。

ReadContentAsXXX()系列方法按照指定格式读取当前元素结点或文本结点的内容，并返回对应类型的值。比如，ReadContentAsInt()表示将当前内容按 int 类型读取，并返回读取到的 int 值。ReadContentAsDateTime()表示将当前内容按 DateTime 类型读取，并返回读取到的 DateTime 类型数据。

ReadElementContentAsXXX()系列方法按照指定格式读取当前元素结点的内容，并返回对应类型的值。比如，ReadElementContentAsInt()表示将当前内容按 int 类型读取，并返回读取到的 int 值。ReadElementContentAsDateTime()表示将当前内容按 DateTime 类型读取，并返回读取到的 DateTime 类型数据。

了解了上面的常用方法，接下来使用 XmlReader 以只读、不缓存、只进的方式读取 XML 文件的数据，通常需要以下几个步骤：

（1）首先通过静态方法 XmlReader.Create()创建一个包含 XML 数据的读取器，即 XmlReader 类实例。

（2）通过 XmlReader.Read()方法读取 XML 数据流的下一个结点，使当前结点向前移动。

（3）通过 XmlReader.NodeType 属性判断当前结点的类型。

（4）根据当前结点的类型和名称对当前结点的数据进行处理。通常需要对 NodeType 为 Element 和 EndElement 的结点进行处理。

（5）读取完成后，通过 XmlReader.Close()方法关闭包含 XML 文件的数据流。

如示例代码 17-2 所示，ReadXmlWithReader()方法演示通过 XmlReader 类读取示例代码 17-1 所示的 UserList.xml 数据。首先，通过 XmlReader.Create()方法创建一个 XmlReader 对象 reader，然后，通过 reader.Read()方法不断在文件前方读取一个结点，直到文件结束（reader.EOF 为 True）。对于每个结点通过 NodeType 属性判断它的类型，如果是 Element，则根据 Name 属性判断是哪个结点，并打印出结点的信息。

示例代码 17-2

```
static void ReadXmlWithReader( )
{
    //用指定文件创建 XmlReader 对象，用于读取 XML 数据
    XmlReader reader = XmlReader.Create(@"UserList.XML");
    //一直读取，直到文件结束
    while (!reader.EOF)
    {
        reader.Read( );           //向前读取一个结点
        if (reader.NodeType == XmlNodeType.Element)
        {
            switch (reader.Name)
            {
                case "UserList":
                    System.Console.Write("用户列表，");
                    reader.MoveToAttribute("Count");
                    System.Console.WriteLine("有{0}个用户：", reader.
                    ReadContentAsInt( ));
                    break;
                case "User":
                    reader.MoveToAttribute("ID");
                    System.Console.Write("编号:{0}, ", reader.
                    ReadContentAsString());
                    break;
                case "Name":
                    System.Console.Write("姓名:{0}, ", reader.
                    ReadElementContentAsString( ));
                    break;
```

```
                case "Age":
                    System.Console.Write("年龄:{0}, ", reader.
                    ReadElementContentAsInt( ));
                    break;
                case "XingBie":
                    System.Console.Write("性别:{0}, ", reader.
                    ReadElementContentAsString( ));
                    break;
                case "Telphone":
                    System.Console.Write("电话:{0}, ", reader.
                    ReadElementContentAsString( ));
                    break;
                case "Email":
                    System.Console.WriteLine("邮箱:{0}", reader.
                    ReadElementContentAsString( ));
                    break;
                default:
                    break;
            }
        }
    }
    reader.Close( );
}
```

示例代码 17-2 的输出如下，从中可以看出，通过 XmlReader.Read()可以读取任何 XML 数据，而且可以将数据按照指定的类型读入。

```
用户列表，有 3 个用户:
编号:1, 姓名:张三, 年龄:18, 性别:男, 电话:13112345678, 邮箱:zhangsan@126.com
编号:2, 姓名:李四, 年龄:25, 性别:男, 电话:13112348888, 邮箱:lisi@126.com
编号:3, 姓名:黄花, 年龄:22, 性别:女, 电话:13112346666, 邮箱:huanghua@126.com
```

技巧：在 XmlReader 第一次创建之后，必需先调用 XmlReader.Read()方法获得第一个结点，然后才开始访问该结点的内容。

17.2.2　用 XmlWriter 保存 XML 数据

XmlWriter 类提供一种快速、非缓存和只进的方式来生成包含 XML 数据的流或文件。可以用于构建符合 W3C 可扩展标记语言建议和 XML 中的命名空间建议的 XML 文档。使用 XmlWriter 以只读、不缓存、只进的方式写入 XML 文件的数据，通常需要以下几个步骤：

（1）通过 XmlWriter.Create()方法创建一个基于文件或数据流的 XmlWriter 对象。

（2）通过 XmlWriter.WriteStartDocument()方法写入 XML 文件的标准头部“<?xml…?>”，如果没有该操作，在 XmlWriter 第一次写入数据时会自动调用该方法写入 XML 文件的头部。

（3）必要时，通过 XmlWriter.WriteComment()方法写入文件描述。

（4）通过 XmlWriter.WriteStartElement()方法写入元素的头。

（5）如果该元素有属性，需要通过 XmlWriter.WriteAttributeString()方法写入该元素的所有属性。

（6）如果该元素需要直接写入值，就通过 XmlWrite.WriteValue()方法写入各种类型的值。

（7）最后通过 XmlWriter.WriteEndElement()方法关闭该元素的写入。

（8）如果元素带有多个子元素，则对每个子元素重复第 4～8 步的写入操作。

（9）通过 XmlWriter.Close()关闭写入器和数据流或文件。

本节通过实例展示如何通过 XmlWriter 类，生成一个类似 Employee.xml 的 XML 文件，首先介绍即将用到的 XmlWriter 的几个常用方法。

- ❑ WriteStartDocument()：该方法用来写入 XML 文件的标准头部，包含文件的 XML 规范版本、命名空间等，格式为“<?xml...?>”。
- ❑ WriteEndDocument()：该方法用来结束整个 XML 文件的写入。
- ❑ WriteStartElement()：该方法开始指定名称的 XML 元素的写入，它具有 3 个重载版本，最常用的 1 个定义如下。
  - ➢ XmlWriter.WriteStartElement(string name)：其中 name 表示要写入元素的限定名。
- ❑ WriteAttributeString()：该方法用在 XmlWriteStartElement()之后，写入当前元素的指定属性，它包含 3 个重载版本，最常用的 1 个定义如下。
  - ➢ XmlWriter.WriteAttributeString(string name, string value)：其中 name 表示要写入属性的名称，value 表示要写入属性的值的字符串形式。
- ❑ WriteValue()：该方法用来在当前位置写入一个各种类型的值，它包含 1 个参数，根据参数类型写入该参数的文本值，它可以支持：int、long、float、double、decimal、bool、Datetime、object 几种类型的写入。
- ❑ WriteEndElement()：该方法用来结束最近一个没有关闭的 WriteStartElement()操作，它必需和 WirteStartElement()方法成对使用。
- ❑ WriteComment()：该方法用来写入指定的 XML 文件注释，格式为“<--!注释文本-->”。
- ❑ WriteWhiteSpace()：该方法用来写入指定的空白字符串，它包含一个 string 类型参数，如果传入字符串不是空字符串，则会抛出一个异常。

如示例代码 17-3 所示，首先，通过 XmlWriter.Create()方法创建一个 XmlWriter 对象 writer，然后通过 WriteStartDocument()和 WriteEndDocument()写入 XML 文档头。通过 WriteStartElement()和 WriteEndElement()写入元素 DogList 和 Dog 元素。并且在需要换行的地方通过 WriteWhitespace()写入换行。

示例代码 17-3

```
static void WriteXmlWithWriter( )
{
    //创建 XmlWriter 对象 writer
    XmlWriter writer = XmlWriter.Create(@"C:\Dogs.xml");
    //写入 XML 文件的头
    writer.WriteStartDocument( );
    writer.WriteWhitespace(System.Environment.NewLine);
    //写入 DogList 元素
    writer.WriteStartElement("DogList");
    writer.WriteWhitespace(System.Environment.NewLine);
```

```
    for (int index = 0; index < 2; index++)
    {
        //写入 Dog 子元素
        writer.WriteStartElement("Dog");
        writer.WriteWhitespace(System.Environment.NewLine);
        //写入 ID 子元素
        writer.WriteStartElement("ID");
        //写入 ID 元素的值
        writer.WriteValue(index);
        writer.WriteEndElement( );
        writer.WriteWhitespace(System.Environment.NewLine);
        writer.WriteEndElement( );
        writer.WriteWhitespace(System.Environment.NewLine);
    }
    writer.WriteEndElement( );
    writer.WriteWhitespace(System.Environment.NewLine);
    writer.WriteEndDocument( );
    writer.Close( );            //关闭 XML 文件
}
```

正常运行示例代码 17-3 之后，可以在路径“C:\”下看到文件 Dogs.XML，打开可以看到如下所示的 XML 数据。

```
<?xml version="1.0" encoding="utf-8"?>
<DogList>
<Dog-0>
<ID>0</ID>
</Dog-0>
<Dog-1>
<ID>1</ID>
</Dog-1>
</DogList>
```

提示： 示例代码 17-3 中，大量的 WriteWhitespace()方法，只是为了写入换行，让 XML 数据更加易读。然而也可以看出 XmlWriter 类在写入 XML 数据时并不方便，17.2.3 节将介绍 DOM 访问 XML 数据。

17.2.3　用 XmlDocument 加载 XML 数据

在.NET 类库中，XmlDocument 类实现 XML 数据在内存中的树状表示形式，它可以加载和保存现有 XML 数据，还允许对 XML 数据进行导航和编辑。XmlDocument 的派生类 XmlDataDocument 还允许将 XML 数据与 DataSet 同步转换，使得数据访问更加灵活。

在 XmlDocument 类中，把 XML 数据中的元素、描述、注释、空白等统称为是 XML 结点，并用类 XmlNode 表示，XmlNode 类是所有 XML 文档结点的基类，它派生了 XmlAttribute、XmlComment、XmlDeclaration 等分别表示特定类型的 XML 文档结点。

一个 XmlNode 可以包含多个 XmlNode 子结点，XmlNode 及其子结点构成一个树状结构。通过 XmlNode 类可以遍历、添加、删除、修改它的子结点，还可以直接定位到第一个和最后一个子结点，定位它的前后结点等。

通过 XmlDocument 类访问 XML 数据，首先需要利用 XmlDocument.Load()和 XmlDocument.LoadXml()方法，从具有 XML 数据的文件或数据流中读取一个完整的 XML

数据，其中 Load()和 LoadXml()方法的详细定义如下。

- Load(string fileName)：从一个 XML 文件中读取完整的 XML 数据到内存，fileName 是 XML 文件的路径。
- Load(Stream sm)：从包含 XML 数据的任意类型数据流读取，sm 是包含 XML 数据的数据流。
- Load(TextReader tr)：从包含 XML 数据的文本数据读取器读取 XML 数据，tr 是包含 XML 数据的文本数据读取器。
- Load(XmlReader xr)：从包含 XML 数据的 XML 数据读取器读取 XML 数据，xr 是包含 XML 数据的 XML 读取器。
- LoadXml(string xml)：从一个包含 XML 数据的字符串片段加载 XML 数据，xml 是包含 XML 数据的字符串。

需要注意的是，XmlDocument 类加载的 XML 数据是一个完整的符合 XML 规范的数据，所以要保证它所加载的数据具有并且只有一个根元素结点，否则会产生异常。

如果仅仅是加载 XML 数据，那么 XmlDocument 类将变得毫无意义，而实际开发中肯定是要对 XML 数据中的各个结点进行遍历，然后才能修改和保存这些数据。事实上，通过 XmlDocument.Load()和 XmlDocument.LoadXml()加载 XML 数据之后，可以通过以下属性访问数据结点。

- DocumentElement：该属性用来获取 XML 文档中的根结点，比如示例代码 17-1 所示 XML 文档中的 UserList 结点。所有 XML 文档中的其他结点操作都从该结点开始，通过它的 ChildNodes 属性逐层访问 XML 文档中的所有子结点。
- ChildNodes：该属性从 XmlNode 类继承得到，返回 XML 文档中的所有最高层结点，通常包括 1 个描述结点（即“<?xml ... ?>”，1 个根结点（即 DocumentElement 属性所指向结点），位于第一层的注释等。用 foreach 关键字可以遍历 ChildNodes 它的所有子结点。

要遍历 XML 文档中的所有结点，则需要对每一个 XmlNode 结点的 ChildNodes 使用 foreach 关键字遍历它所包含的子结点，如此递归下去，所有结点都可以访问完成。

如示例代码 17-4 所示，方法 GoThroughtUserList()首先创建一个 XmlDocument 对象，并通过 Load()方法加载 XML 文件数据 UserList.XML。然后，通过 XmlDocument.DocumentElement 属性获取 XML 数据根结点 UserList。之后，通过 foreach 遍历该结点的 ChildNodes 属性，获取所有的 User 结点。最后，再用 foreach 遍历 User 结点的所有子结点并通过 InnerXml 获取结点的值。

示例代码 17-4

```
static void GoThroughtUserList( )
{
    //创建 XmlDocument 对象 xmlDoc
    XmlDocument xmlDoc = new XmlDocument( );
    //通过 Load()方法加载 XML 文件 UserList.XML
    xmlDoc.Load(@"UserList.XML");
```

```
    //打印根结点的数据
    string strUserList = string.Format("用户列表，共有{0}个用户:",
                              xmlDoc.DocumentElement.Attributes["Count"].
                              Value);
    System.Console.WriteLine(strUserList);
    //遍历根结点 UserList 中所有 User 子结点
    foreach (XmlNode userNode in xmlDoc.DocumentElement.ChildNodes)
    {
        //打印 User 结点的数据，ID 属性
        string userInfo = string.Format("  编号:{0}, ", userNode.
        Attributes["ID"].Value);
        foreach (XmlNode usrInfoNode in userNode.ChildNodes)
        {
            if (usrInfoNode.Name == "Name")            //姓名
            {
                userInfo += string.Format("姓名:{0}, ", usrInfoNode.
                InnerXml);
            }
            else if (usrInfoNode.Name == "Age")         //年龄
            {
                userInfo += string.Format("年龄:{0}, ", usrInfoNode.
                InnerXml);
            }
            else if (usrInfoNode.Name == "XingBie")     //性别
            {
                userInfo += string.Format("性别:{0}, ", usrInfoNode.
                InnerXml);
            }
            else if (usrInfoNode.Name == "Telphone")    //电话
            {
                userInfo += string.Format("电话:{0}, ", usrInfoNode.
                InnerXml);
            }
            else if (usrInfoNode.Name == "Email")       //邮箱
            {
                userInfo += string.Format("邮箱:{0}", usrInfoNode.InnerXml);
            }
        }
        System.Console.WriteLine(userInfo);
    }
}
```

示例代码 17-4 的输出如下，可以看出输出结果与示例代码 17-2 完全一样，但是代码的可读性却比示例代码 17-2 要好得多。而且它还有另外一个好处就是可以修改并保存 XML 数据，17.2.4 节将介绍该内容。

```
用户列表，共有 3 个用户:
  编号:1, 姓名:张三, 年龄:18, 性别:男, 电话:13112345678, 邮箱:zhangsan@126.com
  编号:2, 姓名:李四, 年龄:25, 性别:男, 电话:13112348888, 邮箱:lisi@126.com
  编号:3, 姓名:黄花, 年龄:22, 性别:女, 电话:13112346666, 邮箱:huanghua@126.com
```

技巧： 通常在遍历一个未知结构的 XML 数据时，都需要使用递归的方法进行深层扫描，示例代码 17-4 只是一种简单的应用场景。

17.2.4　用 XmlDocument 创建和保存 XML 数据

前面介绍了通过 XmlDocument 加载 XML 数据到内存，很多时候仅仅是只读方式访问 XML 数据还不够，需要添加、修改、删除 XML 文档中的某些结点。通过 XmlDocument 类可以方便地进行 XML 数据的修改，本节将介绍如何通过 XmlDocument 类修改 XML 结点的值。

通过 XmlDocument 类修改 XML 文档数据，通常需要以下几个主要步骤或其中几个步骤。

（1）获取一个包含 XML 文档数据的 XmlDocument 类对象，通常有两种方法来实现这个功能：

- 通过 17.2.3 节所介绍方法加载已有 XML 数据。
- 通过 XmlDocument 类的构造函数创建不包含任何结点的空对象，常用默认构造函数。

（2）通过 XmlDocument 类的 ChildNodes 和 Item 属性获取某个结点（XmlNode 类型），通过 XmlNode 的 Name、Value、InnerText 等属性修改选中结点的数据。

（3）通过 XmlDocument 类的 CreateElement()和 CreateAttribute()方法，创建新的元素结点和属性结点，并通过 XmlNode 的 Name、Value、InnerText 等属性设置新结点的属性。CreateElement()和 CreateAttribute()的常用定义如下。

- CreateElement(string name)：创建具有指定限定名的元素结点，其中 name 表示元素结点的限定名，返回 XmlElement 类型对象。
- CreateAttribute(string name)：创建具有指定限定名的属性结点，其中 name 表示属性结点的限定名，返回 XmlAttribute 类型对象。

（4）通过 XmlDocument 类的 CreateXmlDeclaration()方法创建一个 XML 文档说明，并通过 XmlDocument.AppendChild()方法添加到 XML 文档中。CreateXmlDeclaration()的定义如下。

- CreateXmlDeclaration(string version, string encoding, string standalone)：创建一个具有指定版本和编码的 XML 文档说明。其中，version 表示版本，encoding 表示 XML 文档的编码格式，默认为 utf-8，standalone 表示是否在 XML 声明上写出独立属性，可选 yes 或 no。

（5）通过 XmlDocument 类的 CreateComment()方法创建一个具有指定文本的 XML 注释，并通过 XmlDocument.AppendChild()方法添加到 XML 文档中。

- CreateComment(string data)：创建包含指定文本的 XML 注释，其中 data 表示注释的文本内容。返回 XmlComment 类型对象。

（6）通过 XmlDocument 类的 Save()方法保存一个 XML 文档数据到文件或数据流，它包含以下重载版本：

- Save(Stream sr)：将内存中的 XML 文档数据保存到指定的数据流，其中，sr 表示

一个特定的可以写入的数据流。

- Save(string filename)：将内存中的 XML 文档数据保存到指定的文件，其中，filename 表示 XML 文件名。
- Save(TextWriter tw)：将内存中的 XML 文档数据保存到指定的文本数据写入器，其中，tw 表示一个文本写入器对象。
- Save(XmlWriter xw)：将内存中的 XML 文档数据保存到指定的 XML 数据写入器，其中，xw 表示一个 XML 数据写入器对象。

示例代码 17-5 是 CreateUserList()的代码片段，首先，创建 XmlDocument 对象 xmlDoc。然后，通过 CreateXXX()方法创建对应的结点和属性，并通过 AppendXXX()方法添加属性和子元素。最后，通过 Save()方法将 XML 数据保存到文件 C:\UserList.XML 中。

示例代码 17-5

```
static void CreateUserList( )
{
    //创建 XmlDocument 对象 xmlDoc
    XmlDocument xmlDoc = new XmlDocument( );
    //创建一个 XML 文档声明，并添加到文档
    XmlDeclaration declare = xmlDoc.CreateXmlDeclaration("1.0", "utf-8",
    "yes");
    xmlDoc.AppendChild(declare);
    //创建并添加 UserList 结点
    XmlElement userListEle = xmlDoc.CreateElement("UserList");
    xmlDoc.AppendChild(userListEle);
    //创建并添加 Count 属性
    XmlAttribute countAttr = xmlDoc.CreateAttribute("Count");
    countAttr.Value = "1";
    userListEle.Attributes.Append(countAttr);
    //创建并添加 User 结点
    XmlElement userEle = xmlDoc.CreateElement("User");
    userListEle.AppendChild(userEle);
    //创建并添加 ID 属性
    XmlAttribute idAttr = xmlDoc.CreateAttribute("ID");
    idAttr.Value = "001";
    userEle.Attributes.Append(idAttr);
    //创建并添加 Name 元素
    XmlElement nameEle = xmlDoc.CreateElement("Name");
    nameEle.InnerText = "李四";
    userEle.AppendChild(nameEle);
    //通过 Save()方法保存数据到 XML 文件 UserList.XML 中
    xmlDoc.Save(@"C:\UserList.XML");
}
```

示例代码 17-5 中，方法 CreateUserList()生成的 UserList.XML 文档内容如下所示。从中可见，通过 XmlDocument.Save()方法保存的 XML 数据，会自动在文件中添加换行和制表符等空白，使得 XML 数据看起来整齐美观。

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<UserList Count="1">
  <User ID="001">
    <Name>李四</Name>
  </User>
</UserList>
```

17.3　了解 LINQ to XML

LINQ to XML 提供使用 LINQ 在内存中操作 XML 数据的编程接口，它使用最新的.NET Framework 语言功能，使得 XML 数据操作更加快速高效，代码更加简洁。本节将介绍 LINQ to XML 的基本原理，基本架构等基础概念。

17.3.1　了解 LINQ to XML

在.NET Framework 4.0 中，增加了 System.Xml.Linq 命名空间，该命名空间提供了一套新的读写 XML 数据的编程接口——LINQ to XML。

LINQ to XML 是一种启用了 LINQ 的内存 XML 编程接口，它让开发人员可以在.NET Framework 编程语言中处理 XML 数据。和文档对象模型（DOM）一样，LINQ to XML 也是将 XML 文档置于内存中，开发人员可以查询、修改 XML 文档，修改之后，还可以将其另存为文件，也可以将其序列化然后通过网络发送。

LINQ to XML 最重要的优势在于它与 Language-Integrated Query（LINQ）的集成，可以对内存 XML 文档编写查询，从而检索元素和属性的集合。LINQ to XML 的查询功能在功能上（尽管不是在语法上）与 XPath 和 XQuery 具有可比性，但是 Visual C# 2008 集成 LINQ 后，可提供更强的类型化功能、编译时检查和改进的调试器支持。

通过将查询结果用作 XElement 和 XAttribute 对象构造函数的参数，实现了一种功能强大的创建 XML 树的方法。这种方法称为“函数构造”，利用这种方法，开发人员可以方便地将 XML 树从一种形状转换为另一种形状。

.NET Framework 4.0 中，通过 System.Xml.Linq 命名空间提供实现 LINQ to XML 的所有类。表 17-2 列出这些类，通过它们可以进行以下 XML 操作：

- ❑ 从文件或流加载 XML。
- ❑ 将 XML 序列化为文件或流。
- ❑ 使用功能构造从头创建 XML 树。
- ❑ 使用 LINQ 查询来查询 XML 树。
- ❑ 操作内存中的 XML 树。
- ❑ 使用 XSD 验证 XML 树。

表 17-2　LINQ to XML主要类

名　　称	说　　明
Extensions	该类提供所有 LINQ to XML 类的扩展方法
XAttribute	该类表示一个 XML 结点的属性（Attribute）
XCData	该类表示一个包含 CDATA 的文本结点，在 XML 文件中格式为：<CDATA>……</CDATA>
XComment	该类表示一个 XML 文件的注释，在 XML 文件中格式为：<!--……-->
XContainer	该类表示 XML 文件中可以包含其他结点的结点

续表

名　　称	说　　明
XElement	该类从 XContainer 类派生，表示一个 XML 元素，在 XML 文件中格式为：<ElementName>……</ElementName>
XDocument	该类从 XContainer 类派生，表示一个完整的 XML 文档
XDocumentType	该类表示 XML 文档的类型定义（DTD）
XDeclaration	该类表示一个 XML 文件的 XML 声明，通常出现在 XML 文件头部，格式为：<xml?……?>
XName	该类表示 XML 文件中元素或属性的名称
XText	该类表示一个文本结点
XNamespace	该类表示一个 XML 命名空间
XNode	该类表示 XML 树中结点的抽象概念，它是以下类型的父类：XContainer、XElement、XComment、XDocument、XProcessingInstruction、Xtext
XNodeDocumentOrderComparer	该类包含用于比较结点的文档顺序的功能
XNodeEqualityComparer	该类用来比较结点以确定其是否相等
XObject	该类用来表示 XML 树中的结点或属性
XObjectChangeEventArgs	该类提供有关 Xobject 类属性的 Changing 和 Changed 事件的数据
XProcessingInstruction	该类用来表示一个 XML 的处理指令
XStreamingElement	该类表示支持延迟流输出的 XML 树中的元素
LoadOptions	该枚举类型指定分析 XML 时的加载选项，包括 None、Preserve Whitespace、SetBaseUri、SetLineInfo 这 4 个可选值
SaveOptions	该枚举指定序列化选项，包括 None、DisableFormatting 两个可选值
XObjectChange	该枚举指定为 XObject 引发事件时的事件类型，包括 Add、Remove、Name、Value 这 4 个可选值

注意：在使用表 17-2 中的类和枚举之前，首先需要引用 System.Xml 和 System.Xml.Linq 命名空间，代码如下：

```
using System.Xml;
using System.Xml.Linq;
```

17.3.2　用 XElement 创建 XML 元素

在 LINQ to XML 中用 XElement 类表示 XML 文档中的元素，XElement 类提供了多个不同版本的构造函数，其中，用于创建简单的单个 XML 元素的重载版本定义如下：

```
public XElement(XName name);
public XElement(XElement element);
public XElement(XName name, object content);
```

其中，name 表示包含元素名称的 XName 对象。element 表示新 XML 元素的源元素，该函数创建一个源元素的深层副本。content 是 object 类型，表示元素的内容，所以它可以是任何数据类型，而显示的内容则是该对象的 ToString()方法所返回的字符串。

值得注意的是，XName 类并没有提供构造函数，所以并不能创建 XName 类型对象作为参数传入 XElement 构造函数。但是存在一个从 string 类型到 XName 类型的隐式转换，

所以只需要将一个 string 类型参数传入到 XElement 构造函数即可。

如示例代码 17-6 中，CreateSimpleXmlElement()方法演示用 XElement 类的多个重载版本创建简单的 XML 元素实例，其中，包括简单的 XML 元素。

示例代码 17-6

```
static void CreateSimpleXmlElement( )
{
    //通过指定元素名的构造函数，创建一个空的元素 UserList
    XElement usrLstEle1 = new XElement("UserList");
    System.Console.WriteLine("usrLstEle1: {0}", usrLstEle1);
    //通过指定元素创建一个相同的元素 UserList
    XElement usrLstEle2 = new XElement(usrLstEle1);
    System.Console.WriteLine("usrLstEle2: {0}", usrLstEle2);
    //通过指定元素数据的构造函数，创建一个指定内容的 User 元素
    XElement userEle1 = new XElement("User", "李四");
    System.Console.WriteLine("userEle1: {0}", userEle1);
    //通过指定元素数据的构造函数，创建一个指定内容的 User 元素,参数是匿名类型
    XElement userEle2 = new XElement("User", new { Name = "王二", Age = "25" });
    System.Console.WriteLine("userEle2: {0}", userEle2);
}
```

示例代码 17-6 的输出如下，其中，usrLstEle1 和 usrLstEle2 只是简单的只有名字的 XML 元素，userEle1 元素的内容文本则是字符串。userEle2 的内容是匿名类型的字符串形式。

```
usrLstEle1: <UserList />
usrLstEle2: <UserList />
userEle1: <User>李四</User>
userEle2: <User>{ Name = 王二, Age = 25 }</User>
```

注意： XElement 类已经重载了 ToString()方法，它返回该元素的文本形式，包括制表符和空白，所以从示例代码 17-6 中可以打印出对应的 XML 文本。

17.3.3　用 XElement 创建 XML 树

在 LINQ to XML 中，使用 XAttribute 类表示一个 XML 元素的属性，它包含一个（名称，值）数据对，用来表示该属性的名称和值。任何 XML 元素（XElement）都包含一个 XAttribute 列表，用来表示它所包含的所有属性，同一个 XML 元素所包含的属性名不能相同。可以通过 XAttribute 类的构造函数创建 XML 属性，包含以下几个重载版本：

```
public XAttribute(XName name, object value)
public XAttribute(XAttribute other)
```

在第一个版本中，参数 name 表示属性（Attribute）的名称，是 XName 类型，同样只需使用字符串参数即可。参数 value 表示属性的值，它是 object 类型，所以属性的值可以是任何类型，它的显示文本则是该 value 的 ToString()所产生的字符串形式。第二个版本用来创建一个已有属性的深层副本，参数 other 表示要被复制的属性对象。

除了属性之外，XML 元素还可以包含一个或多个子元素，通过 XElement 的可变参数版本构造函数创建具有子元素的 XML 元素，当传入参数为 XElement 类型时，该参数则是所创 XML 元素的子元素。

注意： XML 元素中子元素的排列顺序和它们作为参数传入时的顺序相同，当传入的是数组时，数组中元素可以看成从 0 到 N 的顺序展开的多个参数。

示例代码 17-7 演示如何创建一个包含多个属性和子元素的 XML 元素。首先，创建一个 Xelement 数组 usrEleList，其包含 2 个 User 元素，在创建 User 元素时，通过传入 XAttribute 类为它创建 ID 属性。然后，通过传入 XElement 元素到元素 userListEle 中，为 UserList 元素添加子元素。最后，打印出元素 UserList。

示例代码 17-7

```
static void CreateComplexXmlElement( )
{
    //首先创建一组 User 子元素
    XElement[] usrEleList = new XElement[] {
        new XElement("User", new XAttribute("ID", "001"), "张三"),
        new XElement("User", new XAttribute("ID", "002"), "李四"),
    };
    //创建 UserList 元素，并将一组 User 元素作为子元素
    XElement userListEle = new XElement("UserList", usrEleList);
    //添加一个新的 User 子元素
    userListEle.Add(new XElement("User", new XAttribute("ID", "003"),
    "黄花"));
    //添加一个属性 Count 到元素 UserList
    userListEle.Add(new XAttribute("Count", "3"));
    //打印 XML 元素
    System.Console.WriteLine("UserList:\n{0}", userListEle);
}
```

示例代码 17-7 的输出如下，从中可以看出，当 XElement()构造函数中传入参数为 XAttribute 时，该参数则作为 XML 元素的属性添加到元素中，XElement()构造函数传入参数为 XElement 时，该参数作为 XML 元素的子元素添加到元素中。

```
UserList:
<UserList Count="3">
  <User ID="001">张三</User>
  <User ID="002">李四</User>
  <User ID="003">黄花</User>
</UserList>
```

技巧： 除了通过构造函数之外，还可以通过 XElement.Add()方法为 XML 元素添加属性或子元素，正如示例代码 17-7 所示。

17.4　使用 LINQ 查询 XML 元素

LINQ to XML 的核心内容在于查询 XML 元素，17.3 节介绍了如何通过 LINQ to XML 创建完整的 XML 树，本节将全面介绍如何通过 LINQ to XML 查询（包括筛选和排序）XML 树中的元素及其属性。

17.4.1　查询 XML 元素的所有子元素

在前面的章节中讲到，LINQ 查询首先要取得合适的数据源，XElement 类包含了一个 XML 元素所具有的所有属性和子元素。XElement 类提供了 4 个方法来访问当前 XML 元素的属性和子元素，从而为 LINQ 查询提供数据源。

通过 XElement 类的 Elements()方法可以获得当前元素的子元素的集合，将该集合做为数据源，可以进行查询。该方法定义如下：

```
public IEnumerable<XElement> Elements();
public IEnumerable<XElement> Elements(XName name);
```

其中，前者返回当前元素所有的子元素，后者返回当前元素中具有特定名称的子元素，name 表示要查找的子元素名称。

示例代码 17-8 演示如何通过 Elements()方法获取所有子元素集合。首先，通过 Elements()查询所有的子元素，并通过 orderby 子句按照 ID 进行排序。然后，通过 Elements(name)查询所有名为 Man 的子元素。最后，打印出查询结果。

示例代码 17-8

```
static void QueryAllElements( )
{
    //获取用户列表
    XElement userList = CreateUserList( );
    //查询 users 通过 Elements()方法获取数据源，查询所有的用户
    var users =
        from usr in userList.Elements( )
        orderby usr.Attribute("ID").Value
        select usr;
    //打印用户信息
    foreach (var item in users)
        System.Console.WriteLine(item);
    //查询所有的名为 Man 的子元素
    var mans =
        from usr in userList.Elements("Man")
        select usr;
    //打印用户信息
    foreach (var item in mans)
        System.Console.WriteLine(item);
}
static XElement CreateUserList( )
{
    //创建男性用户子元素“李四”
    XElement man1 =
        new XElement("Man",
            new XAttribute("ID", "001"),
            new XAttribute("Age", "22"),
            new XElement("Name", "李四"),
            new XElement("XingBie", "男"),
            new XElement("Phone", "131123456789"),
            new XElement("Email", "LiSi@126.com"));
    //创建男性用户子元素“张三”
    XElement man2 =
```

```
        new XElement("Man",
            new XAttribute("ID", "002"),
            new XAttribute("Age", "23"),
            new XElement("Name", "张三"),
            new XElement("XingBie", "男"),
            new XElement("Phone", "132333456789"),
            new XElement("Email", "Zhangsan@126.com"));
    //创建女性用户子元素“黄花”
    XElement woman1 =
        new XElement("Woman",
            new XAttribute("ID", "003"),
            new XAttribute("Age", "19"),
            new XElement("Name", "黄花"),
            new XElement("XingBie", "女"),
            new XElement("Phone", "130003456789"),
            new XElement("Email", "HuangHua@126.com"));
    //创建用户列表元素
    XElement userList =
        new XElement("UserList",
            new XAttribute("Count", "3"),
            man1, woman1, man2);
    return userList;
}
```

示例代码 17-8 的输出如下，从中可以看出所有成员都按照 ID 从小到大排序，值得注意的是数据源本身并不是按照这个顺序排序的。

```
<Man ID="001" Age="22">
  <Name>李四</Name>
  <XingBie>男</XingBie>
  <Phone>131123456789</Phone>
  <Email>LiSi@126.com</Email>
</Man>
<Man ID="002" Age="23">
  <Name>张三</Name>
  <XingBie>男</XingBie>
  <Phone>132333456789</Phone>
  <Email>Zhangsan@126.com</Email>
</Man>
<Woman ID="003" Age="19">
  <Name>黄花</Name>
  <XingBie>女</XingBie>
  <Phone>130003456789</Phone>
  <Email>HuangHua@126.com</Email>
</Woman>
<Man ID="001" Age="22">
  <Name>李四</Name>
  <XingBie>男</XingBie>
  <Phone>131123456789</Phone>
  <Email>LiSi@126.com</Email>
</Man>
<Man ID="002" Age="23">
  <Name>张三</Name>
  <XingBie>男</XingBie>
  <Phone>132333456789</Phone>
  <Email>Zhangsan@126.com</Email>
</Man>
```

注意：示例代码 17-8 中的 CreateUserList()方法会在后面的章节中广泛使用，其主要用来创建一个 XML 文档数据，作为示例代码访问的数据源。

17.4.2　查询 XML 元素的特定子元素

除了返回所有的子元素外，XElement 类还提供 Element()方法获取当前元素中具有特定名称的子元素，该方法定义如下：

```
public XElement Element(XName name);
```

其中，name 表示要定位的子元素的名称，直接传入 string 类型字符串即可。如果没有指定名称的子元素，则返回 null；如果具有多个指定名称的子元素，则返回文档顺序上的第一个。

示例代码 17-9 演示如何通过 Element()方法，或从 XElement 类中获取某个具有特定名称的子元素，这里同样使用示例代码 17-8 所示的 CreateUserList()方法创建 XML 数据源。

```
static void QueryCertainElement( )
{
    //获取用户列表
    XElement userList = CreateUserList( );
    //获取第一个名为 Man 的元素
    XElement aMan = userList.Element("Man");
    System.Console.WriteLine(aMan);
    //获取第一个名为 Woman 的元素
    XElement aWoman = userList.Element("Woman");
    System.Console.WriteLine(aWoman);
}
```

示例代码 17-9 的输出如下，从中可以看出，aMan 返回了第一个名称为 Man 的子元素，而 aWoman 则返回了第一个名称为 Woman 的子元素。

```
<Man ID="001" Age="22">
  <Name>李四</Name>
  <XingBie>男</XingBie>
  <Phone>131123456789</Phone>
  <Email>LiSi@126.com</Email>
</Man>
<Woman ID="003" Age="19">
  <Name>黄花</Name>
  <XingBie>女</XingBie>
  <Phone>130003456789</Phone>
  <Email>HuangHua@126.com</Email>
</Woman>
```

17.4.3　查询 XML 元素的属性

查询 XML 元素的属性信息是常用的操作之一，在 Xelement 类中，可以通过 Attribute()和 Attributes()方法，获取属性 XElement 的指定属性或全部属性，它们的定义如下：

```
public XAttribute Attribute(   XName name);
```

```
public IEnumerable<XAttribute> Attributes();
public IEnumerable<XAttribute> Attributes(XName name);
```

其中，name 表示属性名称，直接传入 stirng 类型字符串即可。Attributes()方法返回当前元素的所有属性集合。Attributes(name)方法只返回具有特定名称的属性集合，由于 XML 元素的属性具有唯一的名称，所以该方法返回空的或只有一个属性的集合。

示例代码 17-10 演示如何通过 Attributes()和 Attribute()方法，获取 XElement 的元素并通过 LINQ 进行查询，然后打印出查询结果。

示例代码 17-10

```
static void QueryAttributes( )
{
    //获取用户列表
    XElement userList = CreateUserList( );
    //获取第一个名为 Man 的元素
    XElement aMan = userList.Element("Man");
    //查询元素 aMan 的所有属性
    var allAttrs =
        from attr in aMan.Attributes( )
        select attr;
    //打印查询结果
    System.Console.WriteLine("所有属性有：");
    foreach (var item in allAttrs)
    {
        System.Console.WriteLine(item);
    }
    //查询元素 aMan 的名为 ID 的属性
    XAttribute idAttr = aMan.Attribute("ID");
    //打印查询结果
    System.Console.WriteLine("名为 ID 的属性：");
    System.Console.WriteLine(idAttr);
}
```

示例代码 17-10 的输出如下，从中可以看出 Attributes()返回了所有属性，而 Attribute()只返回了一个属性。

```
所有属性有：
ID="001"
Age="22"
名为 ID 的属性：
ID="001"
```

技巧： 当要查找的属性名不存在时，Attribute(name)返回 null，而 Attributes(name)返回一个空的集合，所以如果要进行查询，作者建议使用 Attributes(name)，防止出现异常。

17.4.4　筛选和排序 XML 元素

前面几节介绍了如何通过 XElement 类查询 XML 元素的子元素和属性，上面例子所用到的只是简单的应用场景。通过 LINQ 查询，同样可以对 XML 元素进行复杂的查询，可

以用 where 子句按照属性对 XML 元素进行过滤，也可以用 orderby 子句对查询结果进行排序。

如示例代码 17-11 所示，QueryComplexElement()方法首先通过 CreateUserList()创建一个 XML 文档 userList，然后，查询 oldUsers 以 userList 的所有元素作为数据源，通过 where 子句查询年龄大于 20 的所有用户，并通过 orderby 子句按照年龄逆序排序。查询 anOldUsers 和 oldUsers 类似，但是它按照用户年龄正序排序，并且查询结果为用户姓名、年龄、E-mail 组成的匿名类型。

示例代码 17-11

```
static void QueryComplexElement( )
{
    //获取用户列表
    XElement userList = CreateUserList( );
    //查询年龄大于 20 的用户，并按照年龄从高到低排序
    var oldUsers =
        from user in userList.Elements( )
        where int.Parse(user.Attribute("Age").Value) > 20
        orderby user.Attribute("Age").Value descending
        select user;
    //打印用户信息
    foreach (var item in oldUsers)
        System.Console.WriteLine(item);
    //查询年龄大于 20 的用户，并按照年龄从低到高排序，查询结果元素为匿名类型，包括用户
    的姓名、年龄和电子邮箱
    var anOldUsers =
        from user in userList.Elements( )
        where int.Parse(user.Attribute("Age").Value) > 20
        orderby user.Attribute("Age").Value
        select new { Name = user.Element("Name").Value,
                     Age = user.Attribute("Age").Value,
                     Email = user.Element("Email").Value};
    //打印用户信息
    System.Console.WriteLine( );
    foreach (var item in anOldUsers)
        System.Console.WriteLine(item);
}
```

示例代码 17-11 的输出如下，从中可以看出，通过 LINQ 可以对 XML 数据进行任何形式的查询，只是数据源由 LINQ to XML 相关类提供。除此之外，还可以对元素进行函数运算之后进行过滤，从而实现更复杂条件的筛选。

```
<Man ID="002" Age="23">
  <Name>张三</Name>
  <XingBie>男</XingBie>
  <Phone>132333456789</Phone>
  <Email>Zhangsan@126.com</Email>
</Man>
<Man ID="001" Age="22">
  <Name>李四</Name>
  <XingBie>男</XingBie>
  <Phone>131123456789</Phone>
  <Email>LiSi@126.com</Email>
</Man>
```

```
{ Name = 李四, Age = 22, Email = LiSi@126.com }
{ Name = 张三, Age = 23, Email = Zhangsan@126.com }
```

17.4.5　在上下文中查询 XML 元素

除了对 XML 元素本身进行复杂的查询条件判断外，XML 应用中，经常需要根据所在元素上下文状态进行查询，也就是说，根据 XML 元素的前后元素的信息进行判断，是否要过滤当前元素。XElement 类提供了两个方法用来获取该元素前后的兄弟元素，它们是 ElementsAfterSelf()和 ElementsBeforeSelf()。

XElement 类的 ElementsAfterSelf()方法返回 XML 文档顺序中，当前元素后面的同级元素的集合，集合中元素的顺序按照文档中出现的顺序。包括 2 个重载版本，定义如下：

```
public IEnumerable<XElement> ElementsAfterSelf();
public IEnumerable<XElement> ElementsAfterSelf(XName name);
```

相反，XElement 类的 ElementsBeforeSelf()方法返回 XML 文档顺序中，当前元素前面的统计元素集合，集合中元素的顺序按照文档中出现的顺序。同样包括 2 个重载版本，定义如下：

```
public IEnumerable<XElement> ElementsBeforeSelf();
public IEnumerable<XElement> ElementsBeforeSelf(XName name);
```

其中，name 表示要查找的元素的类型，第 1 个版本返回所有的同级元素，第 2 个版本具有指定名称的同级元素。

在 LINQ 查询中，可以利用上面两个方法对 XML 元素进行上下文相关的条件判断，比如查找某个元素，该元素具有指定名称的前导元素或后缀元素。示例代码 17-12 演示如何使用它们进行上下文查询。查询 oneManUsers 在 where 子句通过 ElementsBeforeSelf()方法查询前面只具有一个名为 Man 的 XML 元素。查询 twoManUsers 利用两个 where 子句，分别通过 ElementsBeforeSelf()和 ElementsAfterSelf()方法，查询前面和后面同时具有一个名为 Man 的 XML 元素的元素。

示例代码 17-12

```
static void QueryInXMLContext( )
{
    //获取用户列表
    XElement userList = CreateUserList( );
    //查询前面具有一个名为 Man 的 XML 元素的 XML 元素
    var oneManUsers =
        from user in userList.Elements( )
        where user.ElementsBeforeSelf("Man").Count( ) == 1
        select new
        {
            Name = user.Element("Name").Value,
            Age = user.Attribute("Age").Value,
            Email = user.Element("Email").Value
        };
    //打印查询结果
    foreach (var item in oneManUsers)
        System.Console.WriteLine(item);
```

```
    //查询前面和后面各具有 1 个名为"Man"的 XML 元素的 XML 元素
    var twoManUsers =
        from user in userList.Elements( )
        where user.ElementsBeforeSelf("Man").Count( ) == 1
        where user.ElementsAfterSelf("Man").Count( ) == 1
        select new
        {
            Name = user.Element("Name").Value,
            Age = user.Attribute("Age").Value,
            Email = user.Element("Email").Value
        };
    //打印查询结果
    System.Console.WriteLine( );
    foreach (var item in twoManUsers)
        System.Console.WriteLine(item);
}
```

示例代码 17-12 的输出如下，从中可以看出通过 ElementsBeforeSelf()和 Elements-AfterSelf()可以轻松对 XML 元素的上下文进行查询。

```
{ Name = 黄花, Age = 19, Email = HuangHua@126.com }
{ Name = 张三, Age = 23, Email = Zhangsan@126.com }

{ Name = 黄花, Age = 19, Email = HuangHua@126.com }
```

注意：本节的很多例子都是直接查询当前 XML 元素，更加复杂的情况下，可以通过多个查询，每个查询都是当前元素或其子元素，然后作为数据源进行二次查询。

17.5　使用 XElement 操作 XML 树

除了查询 XML 树中数据之外，从数据流加载 XML 树，并将 XML 保存到数据流也是常用技术。在内存中添加、编辑、删除 XML 树的结点等操作也是常用技术之一，本节将介绍在 LINQ to XML 中如何实现这些基本 XML 树操作。

17.5.1　从文件加载 XML 元素

本章前面的示例代码中，都是直接在内存中创建 XML 树，这样并不是最常用的办法，因为 XML 数据通常是存储在 XML 文件中。所以，XElement 类提供了 Load()方法，该方法从包含 XML 数据的文件或数据流加载 XML 树到内存。

XElement 类的 Load()方法是一个静态函数，它创建一个 XElement 对象，并从文件或数据流读取 XML 数据到内存对象中。其共包括 6 个重载版本，定义如下：

```
public static XElement Load(string uri);
public static XElement Load(TextReader textReader);
public static XElement Load(XmlReader reader);
public static XElement Load(string uri,LoadOptions options);
public static XElement Load(TextReader textReader,LoadOptions options);
public static XElement Load(XmlReader reader,LoadOptions options);
```

其中，uri 为 string 类型，表示要包含 XML 数据的文件路径，可以是相对路径或绝对路径。参数 textReader 和 reader 表示包含 XML 数据的文本数据流或 XML 数据流。

参数 options 是 LoadOptions 枚举类型，用来指定空白行为及是否加载基 URI 和行信息。LoadOptions 枚举包括以下几个可选值：

- None：不保留无关紧要的空白，也不加载基 URI 和行信息。
- PreserveWhiteSpace：分析时保留无关紧要的空白。
- SetBaseUri：从 XmlReader 请求基 URI 信息，并通过 BaseUri 属性使此信息可用。
- SetLineInfo：从 XmlReader 请求行信息并通过 XObject 上的属性使此信息可用。

此外，XElment 类间接实现了接口 IXmlLineInfo，该接口可以提供它所在的行和列位置信息，主要包括 3 个成员：

- HasLineInfo()：返回 bool 类型的值，表示该对象是否包含行信息。
- LineNumber：返回 int 值，表示当前对象所在文档中的所在行号。
- LinePosition：返回 int 值，表示当前对象在文档中所在行中的具体位置（列号）。

通过 IXmlLineInfo 接口的 LineNumber 和 LinePosition 两个属性的值，可以完全定位当前元素在 XML 文档中的位置，便于跟踪。

示例代码 17-13 中，LoadUserListXML()首先将一段简单的 XML 数据保存到临时文件 userList.xml 中，然后通过 XElement.Load()方法从该文件加载 XML 数据到内存，并通过 options 参数指定需要加载行信息。在最后的 foreach 中，通过将 XElement 对象转换成 IXmlLineInfo 接口，从而获取数据的行号等信息。

示例代码 17-13

```
static void LoadUserListXML( )
{
    //创建一段简单的 XML 数据
    string xmlData =
              @"<UserList>
                  <User> 张三 </User>
                  <User> 李四 </User>
                  <User> 黄花 </User>
                </UserList>";
    //通过 File.WriteAllText()方法将 XML 数据写入到文件 userList.xml 中
    File.WriteAllText("userList.xml", xmlData);
    //通过 XElment.Load()方法从文件加载 XML 数据，并同时加载行和空白信息
    XElement ele = XElement.Load("userList.xml", LoadOptions.SetLineInfo);
    //依次打印 ele 中的所有元素，并判断是否具有行号信息，有则打印
    foreach (XElement item in ele.DescendantsAndSelf( ))
    {
        //将 XElement 强制转换成 IXmlLineInfo 类，从而获取是否包含行号信息
        IXmlLineInfo lineInfo = (IXmlLineInfo) item;
        if (lineInfo.HasLineInfo( ))
            System.Console.Write("Line {0} Position {1}:\t", lineInfo.
            LineNumber, lineInfo.LinePosition);
        else
            System.Console.Write("Line XX :\t");
        System.Console.Write(item.Name);
        System.Console.Write(item.Value);
        System.Console.WriteLine( );
    }
}
```

示例代码 17-13 的输出如下，从中可以看出，XElement.Load()方法成功加载了文件 UserList.xml 中的数据，并且包含了正确的位置信息。

```
Line 1 Position 2:     UserList 张三  李四  黄花
Line 2 Position 28:    User 张三
Line 3 Position 28:    User 李四
Line 4 Position 28:    User 黄花
```

提示：XElment 类间接继承至 XObject 类型，而 XObject 类型实现了 IXmlLineInfo 接口。所有 XElement 类实现了 IXmlLineInfo 接口。

17.5.2　添加任意 XML 子结点

在 XML 文档中，每一个 XML 元素都可以包含一个或多个子元素，因为 XML 文档本身是一种树状的数据组织形式。所以，XElement 类作为 XML 元素在内存中的表现形式，必须支持嵌套的 XML 元素，支持添加或删除 XML 元素。

在 LINQ to XML 中，XElement 类间接地继承至 XNode 类，XNode 类继承至 XContainer 类，XNode 和 Xcontainer 提供了 4 个方法用来添加元素和属性等结点。XElement 类重写了这些方法，间接支持了添加 XML 元素和属性，这 4 个方法如下所示。

- Add()：该方法由 XContainer 类实现，用于在当前结点的子结点的末尾添加内容，被添加的结点作为当前元素的子结点。其包括 2 个重载版本，定义如下，分别用于添加一个或多个内容。

```
public void Add(Object content);
public void Add(params Object[] content);
```

- AddFirst()：该方法由 XContainer 类实现，用于在当前结点的第一个子结点之前添加内容，被添加的结点作为当前元素的子结点。其包括 2 个重载版本，定义如下，分别用于添加一个或多个内容。

```
public void AddFirst(Object content);
public void AddFirst(params Object[] content);
```

- AddAfterSelf()：该方法由 XNode 类实现，用于在当前结点后面添加内容，被添加结点与当前结点同级。其包括 2 个重载版本，定义如下，分别用于添加一个或多个内容。

```
public void AddAfterSelf(Object content);
public void AddAfterSelf (params Object[] content);
```

- AddBeforeSelf()：该方法由 XNode 类实现，用于在当前结点前面添加内容，被添加结点与当前结点同级。包括 2 个重载版本，定义如下，分别用于添加一个或多个内容。

```
public void AddBeforeSelf (Object content);
public void AddBeforeSelf (params Object[] content);
```

值得注意的是，这些方法接受的是 object 和 object[]类型，并非固定为 XElement 类型，也就是说可以添加任何类型的元素到结点。如果参数是 XElement 类型，则添加为 XML 元

素，如果是 Xattribute()类型，则添加为属性，如果是其他类型，则添加为 XML 元素的值。

示例代码 17-14 演示这 4 个方法的具体使用。首先，从 XML 文件 doc.xml 中加载元素 docRoot，并通过 Add()和 AddFirst()分别为它添加两个元素 UserList0 和 UserList2。然后，通过 Add()方法，为 docRoot 添加两个非 XElement 对象，从而演示普通类型添加后作为当前元素的值。最后，通过 AddBeforeSelf()和 AddAfterSelf()方法在元素“张三”前后各添加一个子元素。

示例代码 17-14

```
static void AddElement( )
{
    //创建一段简单的 XML 数据
    string xmlData =
             @"<DocRoot>
                   <UserList1/>
               </DocRoot>";
    //通过 File.WriteAllText()方法将 XML 数据写入到文件 doc.xml 中
    File.WriteAllText("doc.xml", xmlData);
    //通过 XElment.Load()方法从文件加载 XML 数据，并同时加载行和空白信息
    XElement docRoot = XElement.Load("doc.xml", LoadOptions.SetLineInfo);
    //添加两个子元素 UserList0 和 UserList2 到结点 DocRoot 中
    XElement userList0 = new XElement("UserList0");
    XElement userList2 = new XElement("UserList2");
    docRoot.AddFirst(userList0);
    docRoot.Add(userList2);
    //为 DocRoot 添加属性 UserListCount
    docRoot.Add(new XAttribute("UserListCount", 3));
    //为 DocRoot 添加两个普通的元素，作为它的 value
    docRoot.Add("String Value,", 20);
    //为 UserList0 添加 1 个子元素
    XElement zhangSan = new XElement("User", "张三");
    userList0.Add(zhangSan);
    //在用户张三之前和之后各加一个元素
    zhangSan.AddBeforeSelf(new XElement("User", "李四"));
    zhangSan.AddAfterSelf(new XElement("User", "黄花"));
    //打印最后的 DocRoot 元素的内容
    System.Console.WriteLine(docRoot);
}
```

示例代码 17-14 的输出如下，从中可以看出，XAttribute 类型的结点 UserListCount 添加为 DocRoot 的属性（Attribute）。而 XElemnt 类型的结点 UserList0 和 UserList2 添加为 DocRoot 的子元素。普通的类型，比如这里的 stirng 和 int，被添加为 DocRoot 的值。

```
<DocRoot UserListCount="3">
  <UserList0>
    <User>李四</User>
    <User>张三</User>
    <User>黄花</User>
  </UserList0>
  <UserList1 />
  <UserList2 />String Value,20</DocRoot>
```

技巧：可以将 LINQ 查询产生的 IEnumerable<XElement>类型的查询结果直接通过这些方法添加到 XElement 中，也可以将自行创建的 XElement 数组或单个 XElement 对象添加到 XElement 中。

17.5.3　移除全部 XML 子结点

17.5 节介绍了如何添加属性、子元素、值等信息到 XML 元素中，相反地，本节介绍如何从 XElement 类中移除元素。首先，可以通过 XElement 类本身提供的方法移除它的元素和属性，这些方法中最常用的应该是 RemoveAll()和 RemoveAttributs()两个。

RemoveAll()和 RemoveAttributes()方法的定义如下，它们不接受任何参数也不返回任何值，前者用于移除 XElement 中所有的子元素和属性，后者用于移除 XElement 中所有的属性。

```
public void RemoveAll();
public void RemoveAttributes();
```

如示例代码 17-15 中，首先，通过 XElement 类的构造函数创建 UserList 元素，它包含 2 个子元素 User1 和 User2，而且它们都包含自己的属性和子元素。然后，通过 RemoveAttributes()方法移除元素 User1 的所有属性。之后，通过 RemoveAll()方法移除元素 User2 的所有属性和子元素。最后，打印出移除后元素 UserList 的文档数据。

示例代码 17-15

```
static void RemoveAll( )
{
    //首先创建一个 XML 元素 userList
    XElement userList = new XElement("UserList",
              new XAttribute("Count", 3),
              new XElement("User1",
                       new XAttribute("ID", "001"),
                       new XElement("Name", "张三")),
              new XElement("User2",
                       new XAttribute("ID", "001"),
                       new XElement("Name", "李四")));
    //先打印出 userList 的内容
    System.Console.WriteLine("移除前的内容:");
    System.Console.WriteLine(userList);
    //移除 User1 的所有属性
    XElement user1 = userList.Element("User1");
    user1.RemoveAttributes( );
    //移除 User2 的所有属性和子元素
    XElement user2 = userList.Element("User2");
    user2.RemoveAll( );
    //打印出移除后的 UserList 元素的内容
    System.Console.WriteLine("移除后的内容:");
    System.Console.WriteLine(userList);
}
```

示例代码 17-15 的输出如下，从中可以看出，RemoveAttributes()方法只是移除了 XElement 类的所有属性（Attribute），而 RemoveAll()则移除 XElement 类的所有属性和子元素。

```
移除前的内容:
<UserList Count="3">
  <User1 ID="001">
```

```
    <Name>张三</Name>
  </User1>
  <User2 ID="001">
    <Name>李四</Name>
  </User2>
</UserList>
移除后的内容:
<UserList Count="3">
  <User1>
    <Name>张三</Name>
  </User1>
  <User2 />
</UserList>
```

17.5.4　移除部分 XML 子结点

17.5.3 节介绍的方法会一次性移除所有的 XML 元素的所有属性和子元素，但实际开发中，很多时候需要移除 XElement 元素的一个或多个属性和子元素。要做到这一点，需要使用 XElement 类提供的另外两个方法：SetAttributeValue()和 SetElementValue()。

这里读者会觉得奇怪，命名是设置（Set）属性或元素的值，怎么会变成移除呢？SetAttributeValue()用于设置 XElement 元素中指定属性的值，当设置的值为 null 时，表示删除该属性。SetElementValue()用于设置 XElement 元素中指定子元素的值，当设置的值为 null 时，表示删除该子元素。它们的定义如下：

```
public void SetAttributeValue(XName name,Object value);
public void SetElementValue(XName name,Object value);
```

其中，参数 name 表示要设置的属性或子元素的名称，直接传入 string 类型字符串即可，value 表示要设置的新的值，为 null 时表示要删除指定的属性或子元素。当然，如果要设置的属性或子元素不存在，则自动添加到 XElement 元素中。

如示例代码 17-16 演示这两个方法的具体使用，首先，创建 XElement 对象 userList，然后通过 SetAttributeValue()和 SetElementValue()方法移除子元素 User1 和属性 Count。

示例代码 17-16

```
static void RemoveBySet( )
{
    //首先创建一个 XML 元素 userList
    XElement userList = new XElement("UserList",
                new XAttribute("Count", 3),
                new XElement("User1",
                           new XAttribute("ID", "001"),
                           new XElement("Name", "张三")),
                new XElement("User2",
                           new XAttribute("ID", "001"),
                           new XElement("Name", "李四")));
    //先打印出 userList 的内容
    System.Console.WriteLine("移除前的内容:");
    System.Console.WriteLine(userList);
    //移除 User1 的子元素
    userList.SetElementValue("User1", null);
```

```
    //移除 Count 属性
    userList.SetAttributeValue("Count", null);
    //打印出移除后的 UserList 元素的内容
    System.Console.WriteLine("移除后的内容:");
    System.Console.WriteLine(userList);
}
```

示例代码 17-16 的输出如下，从中可以看出，通过 SetAttributeValue()和 SetElementValue()方法都是父结点移除子结点的内容。

```
移除前的内容:
<UserList Count="3">
  <User1 ID="001">
    <Name>张三</Name>
  </User1>
  <User2 ID="001">
    <Name>李四</Name>
  </User2>
</UserList>
移除后的内容:
<UserList>
  <User2 ID="001">
    <Name>李四</Name>
  </User2>
</UserList>
```

17.5.5　XML 子结点自动移除

前面介绍的方法都是通过父元素移除它所包含的属性和子元素，但是有时，子元素或属性本身主动要求从父元素移除，在 LINQ to XML 中可以通过 2 个方法实现这种功能。

- ❑ XNode.Remove()：由 XNode 类实现，XElement 从 XNode 继承而来，用于将当前结点从它的父结点中移除，它的定义如下：

```
public void Remove();
```

- ❑ Extensions.Remove()：LINQ to XML 提供的扩展方法，用于将集合中的所有结点从它们的父结点中移除，包括两个重载版本，定义如下：

```
public static void Remove(this IEnumerable<XAttribute> source);
public static void Remove<T>(this IEnumerable<T> source) where T : XNode;
```

从上面的定义可以看出，前者常用于单个结点要求主动从父结点移除的情形。而后者则通常用于将 LINQ 查询结果（一组 XNode 对象的集合）从它们的父结点移除的情形。

示例代码 17-17 演示这两个 Remove 方法的具体使用。首先，获取一个包含 3 个 User 子元素的 UserList 元素。然后，通过 XNode.Remove()方法，属性 Count 主动从 UserList 中移除。接下来，通过 LINQ 查询产生所有姓李的 User 元素，并通过 Extension.Remove()方法主动从 UserList 中移除。

示例代码 17-17

```
static void AutoRemove( )
{
    //首先创建一个 XML 元素 userList
    XElement userList = new XElement("UserList",
                new XAttribute("Count", 3),
                new XElement("User",
                           new XAttribute("ID", "001"),
                           new XElement("Name", "张三")),
                new XElement("User",
                           new XAttribute("ID", "002"),
                           new XElement("Name", "李四")),
                new XElement("User",
                           new XAttribute("ID", "003"),
                           new XElement("Name", "李花")));
    //先打印出 userList 的内容
    System.Console.WriteLine("移除前的内容:");
    System.Console.WriteLine(userList);
    //主动移除 Count 属性
    XAttribute countAttr = userList.Attribute("Count");
    countAttr.Remove( );
    //查询所有姓李的人，并主动移除
    var liUsers =
        from user in userList.Elements( )
        where user.Element("Name").Value.Contains("李")
        select user;
    liUsers.Remove( );
    //打印出移除后的 UserList 元素的内容
    System.Console.WriteLine("移除后的内容:");
    System.Console.WriteLine(userList);
}
```

示例代码 17-17 的输出如下，从中可以很清楚看到元素 UserList 的 Count 属性通过 Remove()方法移除了，而所有姓李的子元素也通过扩展 Remove()方法主动移除。

```
移除前的内容:
<UserList Count="3">
  <User ID="001">
    <Name>张三</Name>
  </User>
  <User ID="002">
    <Name>李四</Name>
  </User>
  <User ID="003">
    <Name>李花</Name>
  </User>
</UserList>
移除后的内容:
<UserList>
  <User ID="001">
    <Name>张三</Name>
  </User>
</UserList>
```

技巧： IEnumerable<XNode>和 IEnumerable<Xattribute>的扩展方法中，当要移除的属性或子元素来自不同的父结点时，会从各自独立的父结点删除，因此实现同时从多个父结点删除子结点。

17.5.6　保存 XML 元素到文件

前面几节都在介绍如何加载 XML 数据，也介绍了修改 XML 数据，但是修改之后通常需要将最新数据保存到 XML 文件，本节将介绍如何通过 XElement 类保存 XML 数据到文件。

在 LINQ to XML 中保存 XML 数据到文件，只需要通过 XElement 类的 Save()方法即可，该方法具有多个重载版本，可以将 XML 数据保存到文件和流。定义如下：

```
public void Save(string fileName);
public void Save(TextWriter textWriter);
public void Save(XmlWriter writer);
public void Save(string fileName, SaveOptions options);
public void Save(TextWriter textWriter, SaveOptions options);
```

其中，fileName 表示 XML 元素要保存的文件名，textWriter 和 writer 表示 XML 元素保存的文件流，options 参数表示保存 XML 数据时使用的格式，是 SaveOptions 类型，包括如下选项。

- None：保存 XML 数据时按照元素结构进行格式化。
- DisableFormatting：保存 XML 数据时保留所有无关紧要的空白。

示例代码 17-18 演示如何通过 Save()保存 XML 数据到文件。首先创建一个简单的 XML 元素 userList，然后通过 Save()方法用不同的 SaveOptions 进行保存。

示例代码 17-18

```
static void SaveElement( )
{
   //首先创建一个 XML 元素 userList
   XElement userList = new XElement("UserList",
            new XAttribute("Count", 1),
            new XElement("User",
                     new XAttribute("ID", "001"),
                     new XElement("Name", "张三")));
   //保存数据到文件 UserListFormatted.XML
   userList.Save(@"C:\UserListFormatted.XML", SaveOptions.None);
   //保存数据到文件 UserListNone.XML
   userList.Save(@"C:\UserListNone.XML", SaveOptions.DisableFormatting);
}
```

执行完成示例代码 17-18 之后，可以看到用 SaveOptions.None 保存 XML 数据时，生成的 XML 文档具有缩进等对齐格式，更加易读。相反，用 SaveOptions.None 保存的 XML 文档没有任何空白，所有元素都直接连接到一起，不便于阅读。

```
UserListFormatted.xml 的内容:
<?xml version="1.0" encoding="utf-8"?>
<UserList Count="1">
```

```
  <User ID="001">
    <Name>张三</Name>
  </User>
</UserList>

UserListNone.xml 的内容:
<?xml version="1.0" encoding="utf-8"?><UserList Count="1"><User
ID="001"><Name>张三</Name></User></UserList>
```

17.6　小　　结

XML 作为面向对象的扩展标记语言，可以灵活地表示树状结构的数据，被广泛应用于数据存储、Web 开发等领域。本章从 XML 基础知识开始，介绍.NET 中通过 DOM 访问 XML 数据，介绍通过 LINQ to XML 访问 XML 数据。

DOM 是传统的访问 XML 数据的方式，在.NET 中可以通过 XmlDocument 类按照 DOM 模型访问 XML 数据。在.NET 4.0 中，推出了 LINQ to XML，将 LINQ 查询和 XML 数据访问相结合，通过 XElement 类操作 XML 数据。通过本章的学习，读者应该掌握以下知识点：

- 什么是 XML 数据？.NET 支持几种操作 XML 数据的方式？
- 如何通过 XmlReader 读取 XML 数据？
- 如何通过 XmlWriter 将 XML 数据保存到文件？
- 如何通过 XmlDocument 类访问 XML 数据？
- 什么是 LINQ to XML？
- 如何通过 XElement 类创建 XML 元素及其属性和子元素？
- 如何通过 XElement 类访问它的属性和子元素？
- 如何通过 LINQ 查询 XElement 类的属性和子元素？
- 如何通过 LINQ 查询 XElement 类的兄弟元素？
- 如何加载 XML 文件的数据到 XElement 类？
- 如何通过 XElement 类修改 XML 数据？

第 6 篇　项目实战

第 18 章　T-SQL 实例——ATM 交易管理系统

通过前面章节的学习，读者对 SQL Server 2008 的相关知识应该有了一个比较清晰全面的认识，如数据库常用操作、T-SQL 编程、高级特性、存储过程等。在本章中将通过一个实战项目——ATM 交易管理系统，来进一步加深和巩固数据库相关知识。

18.1　ATM 交易管理系统需求分析

衡量软件项目是否成功非常重要的一个因素就是用户需求，所以在开始之前掌握如何分析用户需求是前提条件。用户的需要是什么？他能否把自己的要求描述清楚？该怎么分析这些需求？如何转化为项目功能？需求的确定过程一般从用户提出需求开始，通过直接的面对面沟通，也可以是电话、书信、E-mail 等任何形式的交流，还可以是问卷调查、投票等有趣方式来形成基本文档，然后经过反复分析、论证、商议甚至谈判，最终达成一致，形成最终经过客户确认的需求，也就是客户认可的结果。

需求的收集是一个复杂又难以把握的过程，客户往往不知道自己到底需要一个什么样的软件，并且由于所处立场的不同描述出来的要求也大相径庭。同时，需求又是多变的，由于需求的确定直接关系着软件的成败，所以对于客户和开发团队需求要尽可能的简单易懂，不能有任何歧义，并且需要反复商议确定才能形成，整个需求确定过程如图 18-1 所示。

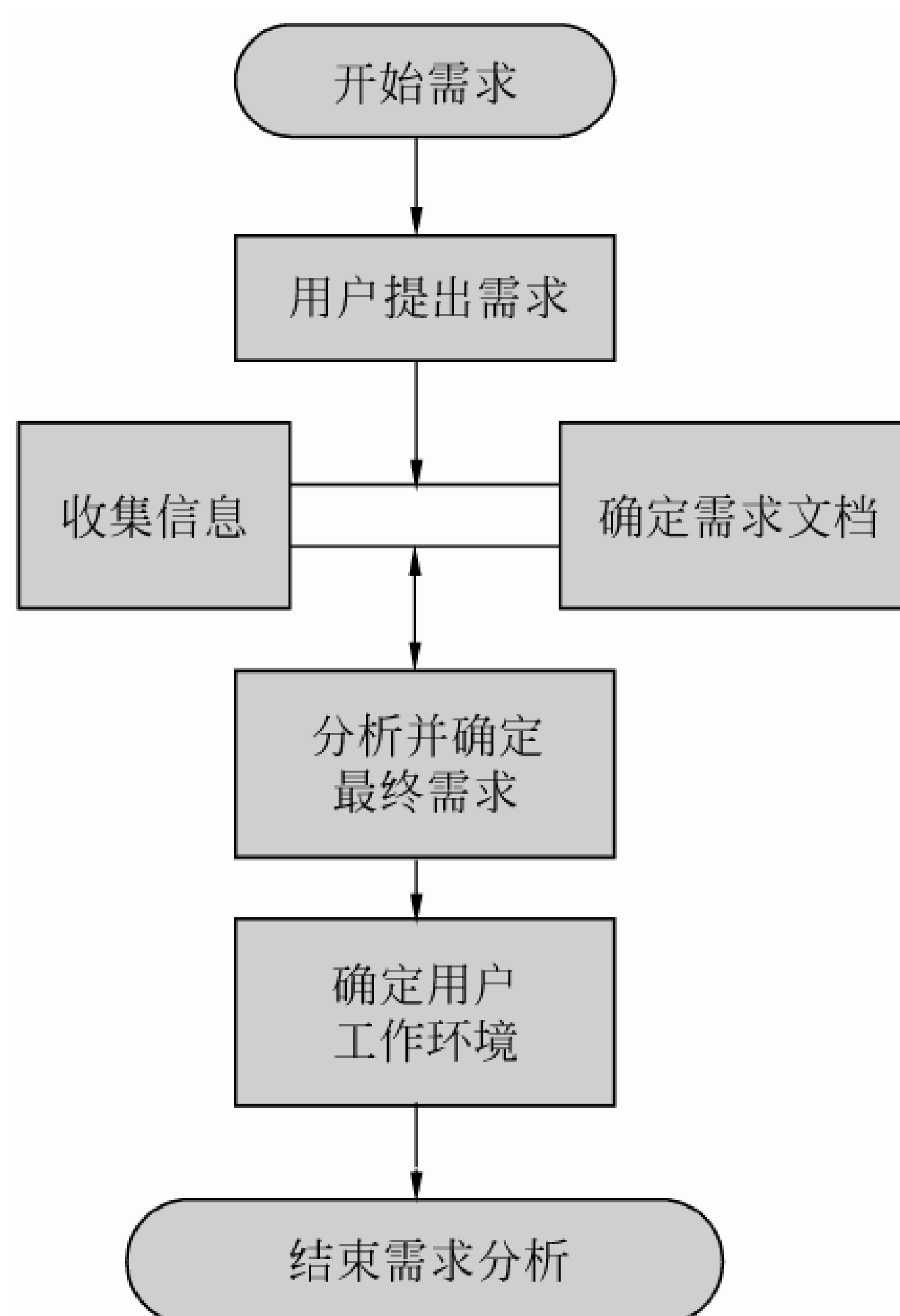


图 18-1　需求确定步骤

18.1.1　分析用户的需求

本例用户是某银行，需要一套 ATM 交易管理系统，经过和客户的初步沟通，对方要求能够实现以下这些功能。

- 新用户能够开卡：按照国家相关规定和银行的需要，用户需提供一些必要信息进行开卡操作，并且存储以备查阅。
- 支取：用户可以随时根据自己的需要，在任意 ATM 提款机上进行支取已存款项操作。
- 存钱：用户可以随时根据自己的需要，在 ATM 提款机上进行存款操作。
- 查询余额：用户可以随时根据自己的需要，在 ATM 提款机上进行余额查询操作。
- 转账：用户能够在 ATM 提款机上进行转账操作，如缴纳手机话费等。
- 明细账：用户可以根据时间区间查询自己的交易历史记录。

需求分析人员需要理清思路，清晰地处理和用户沟通过的每一个细节问题，并将结果记录，为后面论证和确定最终需求做准备。

18.1.2　功能性需求分析

通过 18.1.1 节的论述，大体上掌握了 ATM 提款机交易管理系统的客户要求，接下来需要经过以项目会议的形式，对需求文档进行细化、规范化和可行性分析论证，最终达成一致形成最终需求定稿。确定下来的对于软件功能方面的分析如表 18-1 所示。

表 18-1　系统功能模块说明

功 能 类 别	功能名称、标识符	描　　述
开户操作	客户资料录入	需要正确存储客户详细信息，如姓名、年龄、性别、电话、地址等信息
	开卡操作	按照规则生成唯一卡号、卡上余额、账户类型和客户编号等信息
支取操作	取款	核对卡号密码进行减少余额操作并记录交易信息
存入操作	存款	验证账户和数据合法性进行增加余额操作并记录交易信息
查询操作	查询余额	验证用户名和密码进行余额查询
	查询交易明细	验证用户名和密码按时间进行交易明细查询
转账操作	转账	验证用户名和密码及对方卡号进行复杂的余额更新操作，记录交易详细信息，并保证操作的一致性

18.1.3　系统总用例分析

为了能够把用户的需求描述地更加准确和清晰一些，现在从实际出发，以两种角色为起点，分别用总用例图的形式来说明它们各自的职责和功能。系统分为两种角色的用户，

分别为“操作员”和“普通用户”，其中，操作员角色可以进行开户操作，用户只能进行一些常规业务操作，具体分析如图 18-2 所示。

注意：本章用到的用例图和关系图等全部使用 Microsoft Office Visio 2003 完成，当然能够完成类似功能的软件还有 PowerDesigner、Rational Rose 或 Erwin 等工具。

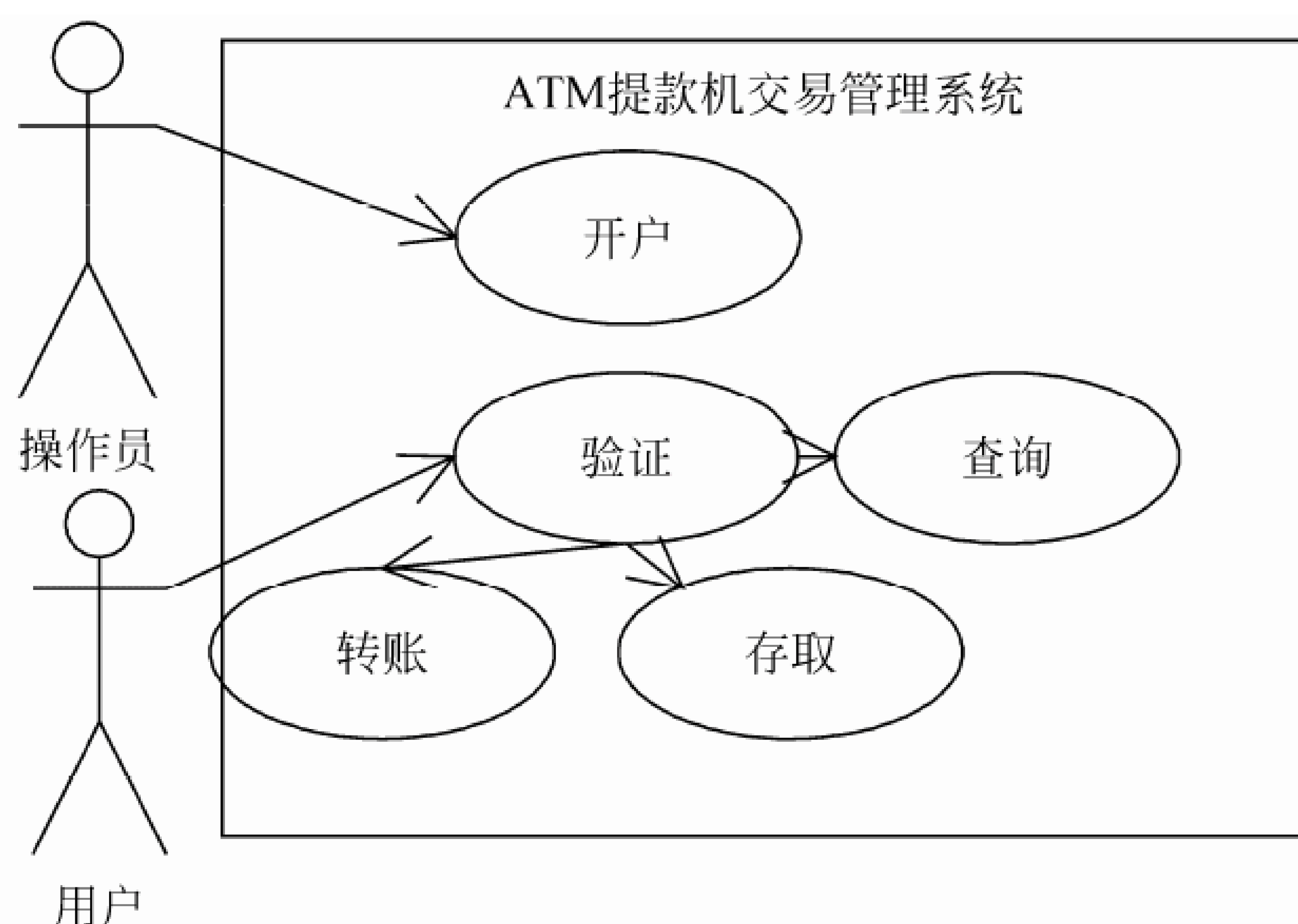


图 18-2　系统总用例图

18.1.4　系统用例分析

【用例 1：开户业务】

❑ 描述：

该模块主要包括针对新用户进行开卡操作，当用户请求使用本系统的时候，操作员需要录入用户基本信息，如姓名、性别、住址、联系电话等信息和开户金额。在信息验证无误的情况下执行新增操作，并同时生成新卡号，提示操作结果。

❑ 参与者：系统操作员。

❑ 用例图：如图 18-3 所示。

【用例 2：支取业务】

❑ 描述：

（1）用户录入卡号和密码以及取款金额，如果验证账户合法性通过，取出相应的金额；

（2）验证不通过提示“账户信息输入有误！”结束本次操作；

（3）如果余额不足提示“账户余额不足！”结束本次操作；

❑ 参与者：用户。

❑ 用例图：如图 18-4 所示。

【用例 3：存款业务】

❑ 描述：

（1）用户输入卡号和金额执行存款操作，成功则提示“存款成功！”；

（2）将本次存款信息记入明细账以备查询；

❑ 参与者：用户。

❑ 用例图：如图 18-5 所示。

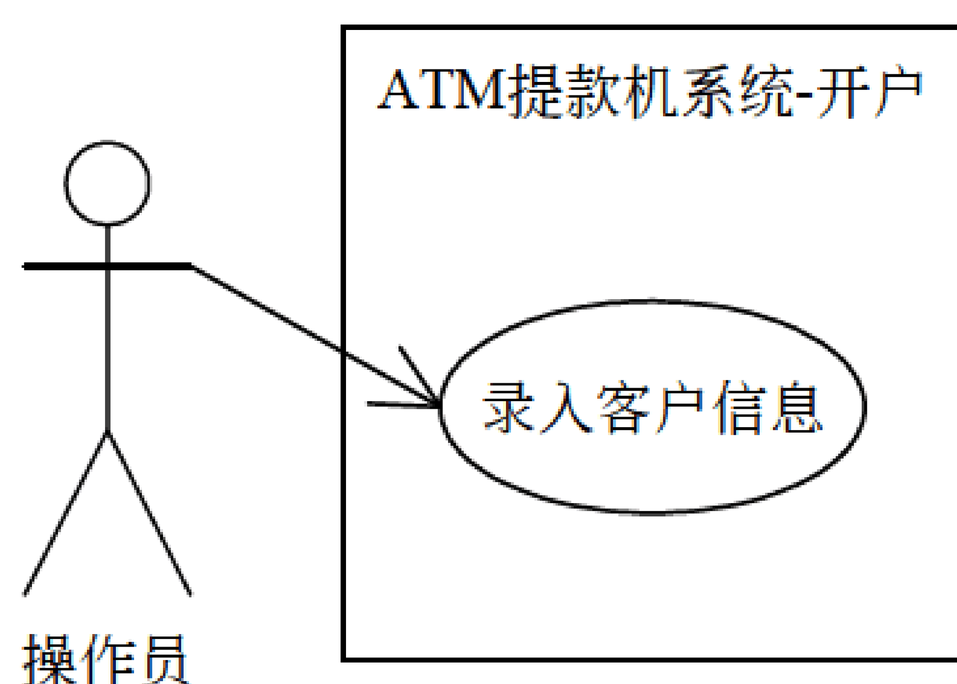


图 18-3　开户用例

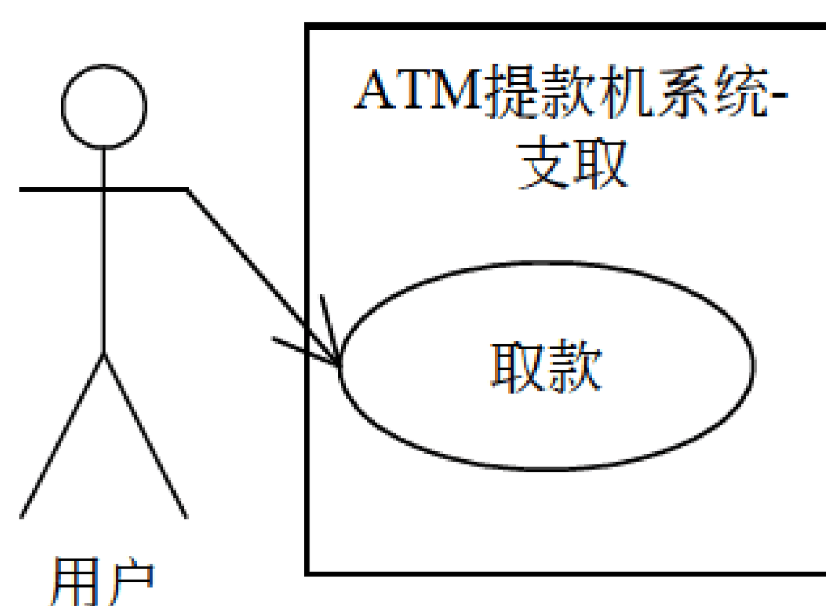


图 18-4　支取用例

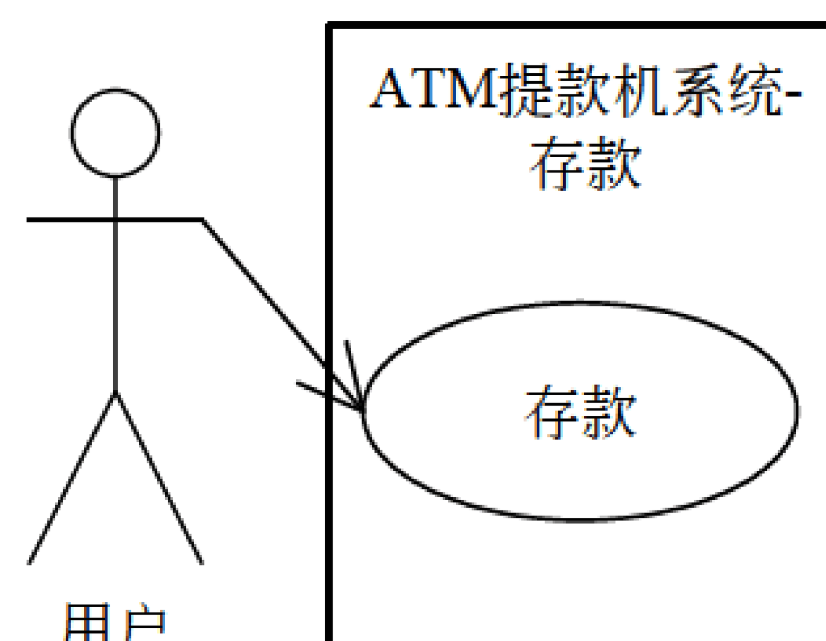


图 18-5　存款用例

【用例 4：查询业务】

❑ 描述：

用户要求查询自己的历史交易记录，可以按照时间段查询，默认时间区间是最近 3 个月。

（1）用户输入卡号和密码进行账户合法性验证，如果验证成功则显示查询结果；

（2）如果验证不通过提示“账户信息输入有误！”结束本次操作；

❑ 参与者：用户。

❑ 用例图：如图 18-6 所示。

【用例 5：转账业务】

❑ 描述：

（1）用户录入卡号和密码及转账金额和对方卡号；

（2）验证通过则从自己账户上减去相应的金额增加到对方账户上（要保证操作的完整性）；

（3）记录明细账，提示“转账成功！”；

（4）验证不通过提示“账户信息输入有误！”结束本次操作；

（5）如果账户余额不足本次操作，则提示“账户余额不足！”结束本次操作；

❑ 参与者：用户。

❑ 用例图：如图 18-7 所示。

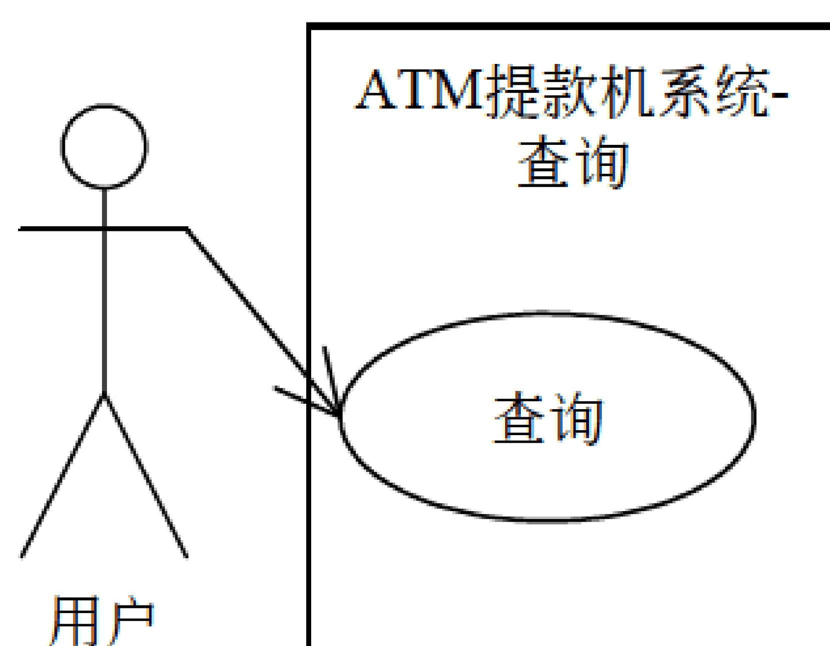


图 18-6　查询用例

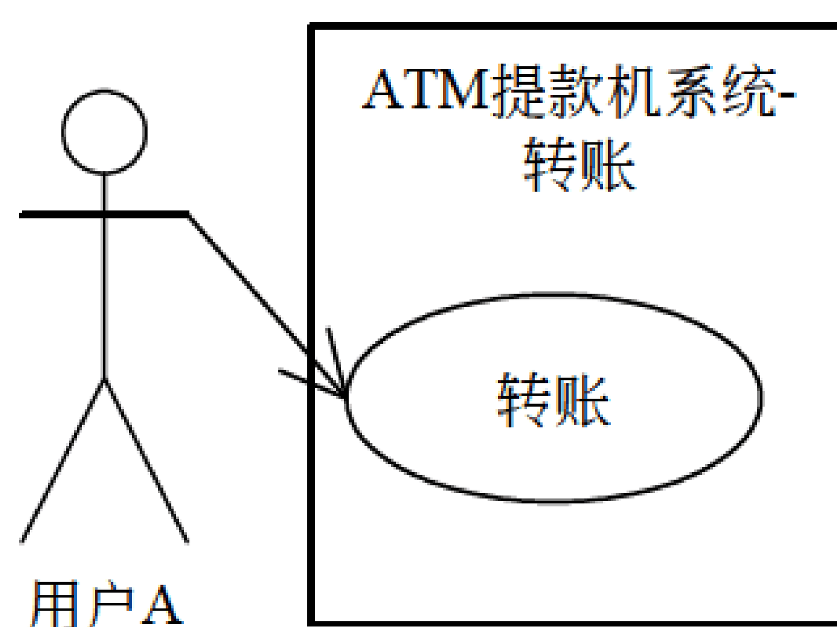


图 18-7　转账用例

18.2　ATM 交易管理系统数据库设计

通过上面对客户信息的收集，整个系统需求基本可以确定，接下来就要分析整个系统

中需要管理的对象、属性及对象间的关系。按照数据库设计的一般步骤：收集客户信息，标识系统要管理的对象，分析各个对象需要关注的属性，标识对象间联系和依赖性来完成整个数据库设计。

18.2.1　实体关系图（E - R 图）

本系统涉及到的管理对象有 3 个：用户、卡和交易记录，其中用户和卡之间是一对多的关系，意思是一个用户可以开多个卡号；卡和交易记录之间是一对多的关系，意思是一张卡可以产生多条交易记录，每个对象需要关注的重点属性如图 18-8 所示。

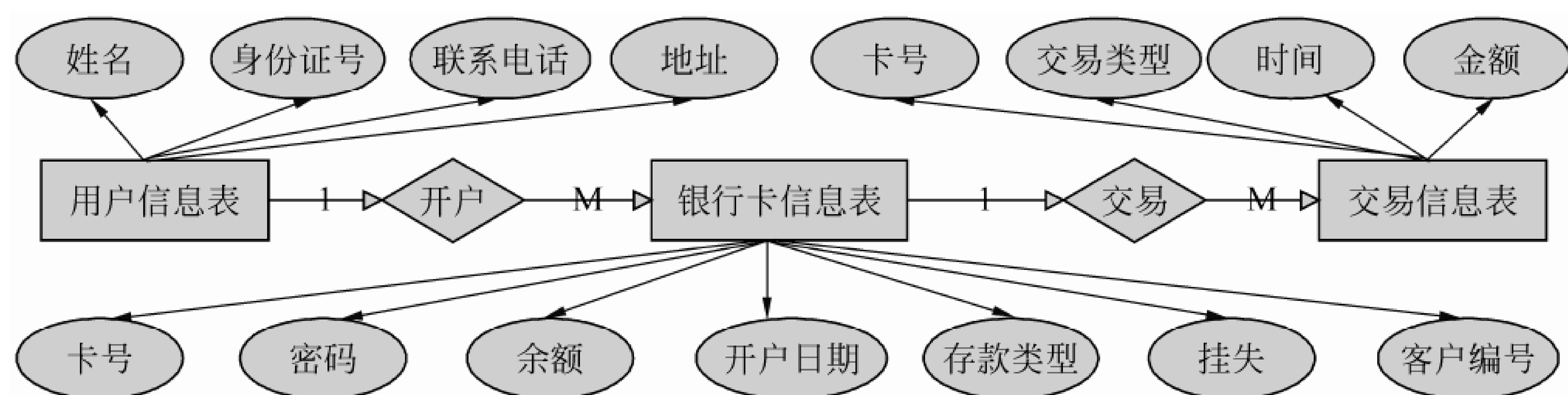


图 18-8　实体关系图

18.2.2　逻辑设计

在明确系统管理对象之后，需要进一步分析确定，才能进入逻辑设计阶段，为了确保数据库设计的合理性、消除数据库设计的冗余、消除更新异常、插入异常和删除异常，需要对整个数据库关系图用《数据库设计范式》进行规范。数据库范式如：

- 第一范式（1NF）无重复的列；
- 第二范式（2NF）属性完全依赖于主键[消除部分子函数依赖]；
- 第三范式（3NF）属性不依赖于其他非主属性[消除传递依赖]。

经过分析，用户信息表中所有字段都和主键的关系非常直接，卡号信息表中的字段也不依赖其他列传递关系，交易记录表就是为了消除冗余拆分出来的表，经分析满足三大范式要求，设计出来的逻辑视图如图 18-9 所示。

但是在任何关系型数据库中，为了达到合理设计的目的，一般都用三大范式来衡量，解释如下。

（1）第一范式可以理解为每列的数据都是不能再次分割存储的，否则就失去该列的意义，也不能有多个值，如果出现重复的属性，就需要新建一个实体来描述。如年龄为 18 岁，存储的时候，一定会存为 18 这个整体，而不是分开存储为 2 列“1”和“8”。

（2）第二范式可以理解为在第一范式的基础上要求更加严格了一些，被描述的实体其他属性都应该和主属性，也就是主键存在依赖关系，都是来描述主属性唯一确定的一个实体的。比如要描述一个学生管理系统，把需要管理的信息放到一个表中（学号，学生姓名、年龄、性别、课程、课程学分、系别、学科成绩，系办地址、系办电话）就存在如下的依赖关系：（学号）→（姓名，年龄，性别，系别，系办地址、系办电话）、（课程名

称）→（学分)、（学号，课程）→（学科成绩）课程学分和学号是没有依赖关系的，就会导致数据冗余：同一门课程由 n 个学生选修，学分就重复 n–1 次；同一个学生选修了 m 门课程，姓名和年龄就重复了 m-1 次。

（3）第三范式要求一个数据库表中不包含在其他表中已包含的非主关键字信息，意思是该实体的所有属性要和主属性存在一个直接的关系，不能因通过任何列的传递而产生关系。一旦发现有和主属性没有直接关系的列，就需要拆分形成一个新的实体。例如，存在一个部门信息表，其中每个部门有部门编号、部门名称、部门简介等信息。那么员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。

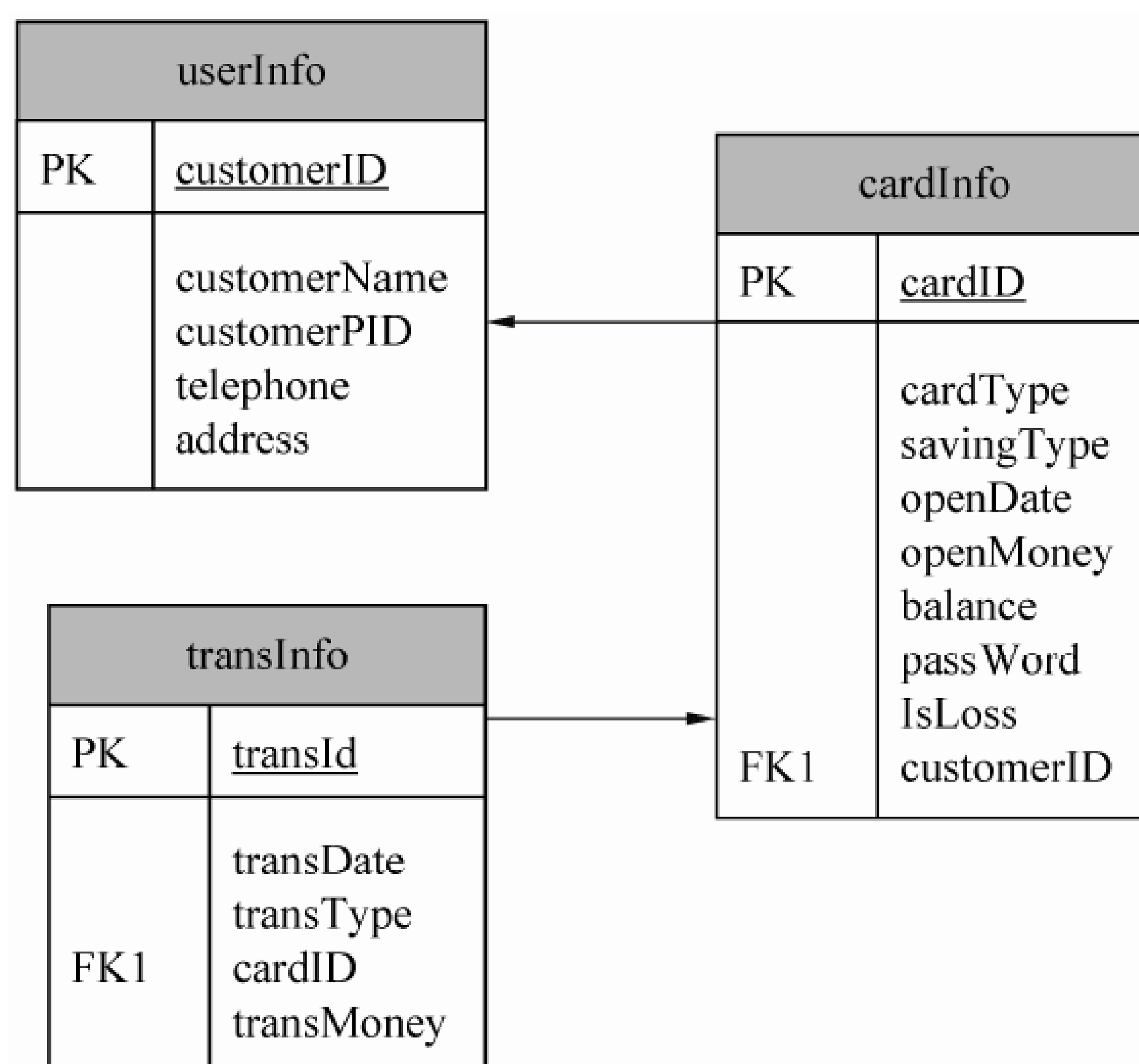


图 18-9　数据库逻辑设计

18.2.3　表设计

由于本案例采用的是 SQL Server 2008 数据库，就需要按照 SQL Server 2008 的数据类型和规范把 18.2.2 节的逻辑设计反映成为物理设计，需要确定每列的数据类型、长度和约束。这里暂定数据库名为 BankMIS，一共需要创建 3 张表，每张表及列的详细信息如表 18-2 所示。

表 18-2　ATM交易管理系统数据库BankDB的表定义

属性 / 表名	字　段	类　型	描　述
userInfo（用户信息表）	customerID	int	客户唯一编号，自增，主键
	customerName	nvarchar(10)	客户姓名，必填项
	customerPID	nvarchar(18)	身份证，必填，兼容15位和18位身份证号，唯一
	telephone	nvarchar(13)	联系电话，必填，可填写手机和固定电话
	address	nvarchar(50)	客户地址，必填

续表

表名＼属性	字段	类型	描　　述
cardInfo（卡信息表）	cardID	char(19)	银行卡号，必填
	cardType	nvarchar (5)	币种，默认为RMB
	savingType	nvarchar (8)	存款类型，必填，如活期、定期
	openDate	datetime	开户日期，必填，默认为当天
	openMoney	money	开户金额，必填，必须大于1
	balance	money	账户余额，必填，必须大于1
	passWord	char(6)	密码，必填，默认为888888
	isLoss	bit	是否挂失，默认为“0”
	customerID	int	客户编号，外键
transInfo（交易信息表）	transId	int	交易明细编号，自增，主键
	transDate	datetime	交易日期，默认当天
	transType	char(4)	交易类型，非空，如支出、存入
	cardID	char(19)	银行卡号，外键
	transMoney	money	交易金额，非空，大于1

注意：这里的数据类型和长度是根据实际调查和与客户沟通确定的，包括专有名词也来源于现实中，再一次说明了需求的重要性。

18.2.4　表实现

完成表设计之后就可以建立数据库了，这个阶段需要使用 SQL Server Management Studio 的查询分析器通过编写 T-SQL 语句的方式完成整个物理实现。建议的操作是：建库、建表、加约束分开操作，这样有利于团队协作开发和系统的维护，现实公司中采取的方式一定是简单有效、易懂并且不容易产生歧义的。

注意：完成整个建表操作也可以通过界面操作的方式，但是一般不建议这样做，像一些数据库如 MySQL、Oracle 等的命令行操作机会明显大一些，而且很多特殊的功能界面并不支持。所以对于数据库的操作建议使用命令方式，刚开始可能觉得不顺畅，时间长了自然就适应了。

（1）建库：设定数据库的初始大小为 3M，增长率为 15%，日志文件和主文件一样，如示例代码 18-1 所示。

示例代码 18-1

```
use master                              --指定初始使用系统数据库
go
```

```
--检测名为 BankMIS 的数据库是否存在，查询系统表
if exists(select * from sysdatabases where name='BankMIS')
 drop database BankMIS                          --如果存在则删除
go                                              --批处理结束标志
create database BankMIS
on primary
(
  name='BankMIS',                               --数据库逻辑名
  filename='F:\data\BankMIS_data.mdf',          --数据库文件名，包含路径
  filegrowth=15%,                               --文件增长率
  size=3                                        --初始大小
)
log on
(
  name='BankMIS_log',                           --日志文件逻辑名
  filename='F:\data\BankMIS_log.ldf',           --日志文件物理名
  filegrowth=5%,                                --日志文件增长率
  size=3                                        --日志文件初始大小
)
go
```

注意：这里创建数据库的时候 F 盘的文件夹最好存在，当然也可以使用在查询分析器里面执行 DOS 命令来创建，但是这种做法极不安全，并且要改变整个服务器的设置，所以不推荐使用。

（2）建表：根据表 18-2 的分析，建立表的 SQL 语句如示例代码 18-2 所示。

示例代码 18-2

```
use BankMIS
go
--创建用户信息表
create table userInfo
(
  customerID int identity(1,1),            --客户编号自增，种子是，增量也是
  customerName nvarchar(10) not null,      --客户姓名
  customerPID nvarchar(18) not null,       --身份证号
  telephone nvarchar(13) not null,         --电话
  address nvarchar(50) not null            --地址
)
--创建卡信息表
create table cardInfo
(
  cardID char(19) not null,                --卡号
  cardType nvarchar(5) not null,           --卡类型
  savingType nvarchar(8) not null,         --存款类型
  openDate datetime not null,              --开户日期
  openMoney money,                         --开户金额
  balance   money,                         --余额
  passWord  char(6) not null,              --密码
  isLossbit,                               --是否挂失
  customerID int                           --客户编号
)
--创建交易信息表
```

```
create table transInfo
(
  transId int identity(1,1),              --交易明细号
  transDate datetime not null,            --交易日期
  transType char(4),                      --交易类型
  cardID char(19),                        --卡号
  transMoney money                        --交易金额
)
go
```

🔔**注意：**上述 SQL 语句中出现的 go 关键字，用于一批次语句执行后，属于批处理标志，一般对执行顺序有要求的时候使用。

（3）增加约束：数据表创建完毕不能说整个数据库建立工作就结束了，为了保证数据的完整性和准确性一般都是要对数据进行约束的，比如年龄就不能是负数，也不能很大等，这里就根据表 18-2 的分析对字段增加约束，确保数据合法性。常用的约束有检查约束、主键约束、外键约束、默认和唯一约束。建立约束代码如示例代码 18-3 所示，

示例代码 18-3

```
--为 userInfo 表增加约束
alter table userInfo
    --为 userInfo 表 customerID 列增加主键约束
    add constraint PK_ID primary key(customerID),
    --为 customerPID 列增加检查约束，用到系统函数 len()
    constraint CK_PID check(len(customerPID)=15 or len(customerPID)=18),
    --确保身份证列数据唯一
    constraint UQ_PID UNIQUE(customerPID),
    --约束电话号码为-88888888 或者 029-88888888 或者 13088888888
    constraint CK_TELEPHONE CHECK( telephone like '0[0-9][0-9][0-9]-[1-9]
     [0-9][0-9][0-9][0-9][0-9][0-9][0-9]' or telephone like '0[0-9][0-9]-
     [1-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' or telephone like '1[3,5,8]
     [0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' )
     go
--为 cardInfo 表增加约束
alter table cardInfo
    add constraint PK_cardID  PRIMARY KEY(cardID),
    constraint CK_CARDID  CHECK(cardID LIKE '4213 4942 [0-9][0-9][0-9][0-9]
    [0-9][0-9][0-9][0-9]'),
    --增加默认约束，默认值为"RMB"
    constraint DF_CARDTYPE  DEFAULT('RMB') FOR cardType,
    --增加检查约束，用 IN 限制可以输入的数据
    constraint CK_SAVINGTYPE  CHECK(savingType IN ('活期','定活两便','定期
    ')),
    --调用日期函数获取当前时间设为默认约束
    constraint DF_OPENTYPE  DEFAULT(getdate()) FOR openDate,
    constraint CK_OPENMONEY  CHECK(openMoney>1),
    constraint CK_BALANCE  CHECK(balance>1),
    constraint CK_PASS  CHECK(passWord LIKE '[0-9][0-9][0-9] [0-9][0-9]
     [0-9]'),
    --设定密码的默认值为"888888"
    constraint DF_PASS  DEFAULT('888888') FOR passWord,
    constraint DF_LOSS DEFAULT(0) FOR isLoss,
    --增加外键约束，cardInfo 表的 customerID 受 userInfo 表的 customerID 字段约束
    constraint FK_customerID FOREIGN KEY(customerID) REFERENCES
```

```
    userInfo(customerID)
go
--为 transInfo 表增加约束
alter table transInfo
    add constraint PK_TID primary key(transId),
    --增加检查约束，设置该列的值只能是"存入","支出"
    constraint CK TRANSTYPE  CHECK(transType IN ('存入','支出')),
    constraint CK TRANSMONEY  CHECK(transMoney>1),
    constraint DF TRANSDATE DEFAULT(getdate()) FOR transDate,
    --增加外键约束，transInfo 表的 cardID 受 cardInfo 的 cardID 列约束
    constraint FK CARDID  FOREIGN KEY(cardID) REFERENCES cardInfo(cardID)
go
```

注意： 增加外键约束的时候是给子表加，也就是给被约束的表加，增加外键约束的前提是主动约束列必须为主表的主键，顺序不能反。

（4）插入测试数据检验数据库表和约束创建的正确与否，最合适、简便的办法就是插入一些测试数据，这里随机抽取 2 条数据，用编写 SQL 语句的方式执行插入操作，代码如示例代码 18-4 所示。

示例代码 18-4

```
---插入测试数据，由于开户就是开卡操作，所以也要插入卡信息表，同时开卡必须存钱，又涉及明细账记账操作
--丹尼尔开户，身份证：，电话：-88888888，地址：北京海淀   开户金额：活期  卡号：4942 1234 5678
insert into userInfo(customerName,customerPID,telephone,address )
    values('丹尼尔','123456789012345','010-88888888','北京海淀')
--开卡操作
insert into cardInfo(cardID,savingType,openMoney,balance,customerID)
    values('4213 4942 1234 5678','活期',1000,1000,1)
--记录明细账操作
insert into transInfo(transType, cardID, transMoney)
    values('存入','4213 4942 1234 5678',1000)
--华盛顿开户，身份证：，电话：，  开户金额：10  定期卡号：4942 1212 1134
insert into userInfo(customerName,customerPID,telephone,address)
    values('华盛顿','335588997711662233','13288888888','美国旧金山')
insert into cardInfo(cardID,savingType,openMoney,balance,customerID)
    values('4213 4942 1212 1134','定期',10,10,2)
insert into transInfo(transType, cardID, transMoney)
    values('存入','4213 4942 1212 1134',10)
go
--查看插入结果
select * from userInfo
select * from cardInfo
select * from transInfo
```

如果最后 3 条查询语句执行结果的数据准确，说明上述建表操作是正确的。在执行测试数据插入时，用到一些业务，比如要开户就必须先插入客户信息，而开卡的同时又存款，存款就必然引发明细账记账操作，这是一个不可分割的完整业务，至于如何保证一系列操作完整性，后面的章节会讲到。

18.3　ATM 交易管理系统常规业务处理

18.2 节完成了数据库创建工作，并且通过插入测试数据的方式验证了数据库的完整性，当然，建立数据库的目的是为了处理业务，是为业务服务。本节就通过一些常规业务的处理再次理解业务和操作间的关系，常规业务有：修改密码操作、银行账务相关的统计操作等，这些全部要使用数据库相关操作来完成。

18.3.1　常规业务部分

所有软件系统都是为业务服务的，业务决定软件的功能，按照前面了解到的客户需求，ATM 交易管理系统必然存在着一些常规业务，如修改密码和挂失等，这些是一般管理类系统通用的功能。下面就针对每个具体的业务进行分析并给出示例代码。

（1）修改密码操作：用户修改密码属于最基本的业务，银行获取到的信息是客户的身份证号和银行卡号，要根据身份验证的结果来执行修改操作。代码如示例代码 18-5 所示。

示例代码 18-5

```
---修改密码
update cardInfo set passWord='123456' WHERE cardID='4213 4942 1212 1134'
select * from cardInfo
```

（2）账号挂失：客户由于不慎将卡丢失，银行应该提供卡挂失功能，卡信息表中的 isLoss 字段就是标志卡是否挂失的，其值为 0 表示正常，其值为 1 表示此卡挂失，不能使用。挂失业务其实就是将卡对应的 isLoss 值修改为 1，但是挂失时，客户有可能说不出自己的卡号，这种情况就需要根据身份证号来执行挂失操作，已知条件为客户身份证号，对应代码如示例代码 18-6 所示。

示例代码 18-6

```
---卡挂失
update cardInfo set isLoss=1 WHERE customerID=(select customerID from
userInfo where customerPID='335588997711662233')
select * from cardInfo
```

注意：这里用到子查询，当子查询的前面为比较运算符时，要确保子查询的结果只有一个才能正确执行。另外，子查询一般都要用小括弧括起来，执行优先级高于父查询。

（3）查询余额 0～5000 之间的定期卡号，显示该卡相关信息：筛选条件有 2 个，余额介于 0～5000 并且存款类型为定期，重点在于两个条件之间关系的处理。代码如示例代码 18-7 所示。

示例代码 18-7

```
---查询余额 0~5000 之间的定期卡号，显示该卡相关信息
select * from cardInfo,userInfo where (cardInfo.customerID=userInfo.
customerID) and (balance between 0 and 5000) and (savingType='定期')
```

注意：这里既用到联合查询又用到复杂的条件组合，一般为了条件的逻辑关系明确起见，通常用小括弧提高条件的优先级，增强程序的可读性。

（4）查询本月开户数量，代码如示例代码 18-8 所示。

示例代码 18-8

```
---查询本月开户人数
select count(*) from cardInfo where  datediff(month,openDate,getdate())=0
```

技巧：datediff 函数的原型是 DATEDIFF (datepart , startdate , enddate) 参数的含义是：enddate 减去 startdate。如果 startdate 晚于 enddate，则返回负值，如果起始日期和终止日期的指定部分一致则返回 0。

（5）存款业务：存款涉及 2 张表的操作，更新卡信息表的余额，插入明细账交易记录。代码如示例代码 18-9 所示。

示例代码 18-9

```
---存款业务：丹尼尔存入 6000 元
update cardInfo set balance=balance+6000 where cardID='4213 4942 1234 5678'
insert into transInfo(transType, cardID, transMoney)values('存入','4213
4942 1234 5678',6000)
select * from cardInfo where cardID='4213 4942 1234 5678'
```

注意：在实时系统中类似余额更新类查询操作，一般采取的做法是 set balance=balance+6000，而不是 set balance=7000，这样是为防止在更新提交前并发一个新的更新。

（6）查询挂失客户的详细资料：客户通过电话挂失后在解挂时需要核对客户的详细资料，根据客户提供的身份证信息，由银行核对其他信息，如地址等，代码如示例代码 18-10 所示。

示例代码 18-10

```
---查询挂失客户的详细信息
select * from userInfo
inner join cardInfo
on userInfo.customerID=cardInfo.customerID
where isLoss=1 and customerPID='335588997711662233'
```

注意：这里用的是内连接，inner join 后面跟要连接的表名，on 后面跟两张表间关系，如果还有筛选条件的话，后面用 where 过滤。

18.3.2　视图和索引部分

从用户角度来看，一个视图是从一个特定的角度查看数据库中的数据。从数据库系统内部来看，一个视图是由 SELECT 语句组成的查询定义的虚拟表。从数据库系统内部来看，视图是由一张或多张表中的数据组成的，从数据库系统外部来看，视图就如同一张表一样，对表能够进行的一般操作都可以应用于视图，例如查询、插入、修改、删除操作等。

在开发中如果能够灵活地使用试图，将会轻而易举地解决一些难题，如简化开发复杂度，定制数据和安全控制等，在本项目中为了给客户一个友好的查询界面，同时简化数据库操作，推荐使用试图，如查询某个客户的具体信息等。

（1）查询客户具体信息视图，代码如示例代码 18-11 所示。

需要把结果以汉字列名的形式显示出来，并隐藏不必要显示的列，创建视图代码如：

示例代码 18-11

```
--查询客户具体信息视图
if exists(select * from sysobjects where name='view_customerInfo')
drop view view_customerInfo
go
create view view_customerInfo
as
select customerName as '客户姓名',customerPID as '身份证号',telephone as
'联系电话',address as '联系地址',cardID as '卡号',cardType as '币种',savingType
as '存款类型',openDate as '开户日期'
from userInfo
inner join cardInfo
on userInfo.customerID=cardInfo.customerID
go
--测试视图语句
select * from view_customerInfo
```

注意：安全创建试图和建库一样需要检测视图是否存在，如果旧的视图存在就删除掉，创建新视图，查询的是系统表 sysobjects，像表、存储过程、视图等都可以在这张表中查到具体信息。

（2）查询交易信息视图，代码如示例代码 18-12 所示。

示例代码 18-12

```
--查询本月交易记录信息视图
if exists(select * from sysobjects where name='view_transInfo')
drop view view_transInfo
go
create view view_transInfo
as
select cardID as '卡号',transDate as '交易时间',transType as '交易类型',
transMoney as '交易金额'
from transinfo
go
--测试
select * from view_transInfo
```

注意：通过这两个视图的创建可以发现，视图中写的是一个没有逻辑语句的查询语句。

（3）创建索引。

使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构，例如 employee 表的姓（name）列。如果要按姓查找特定职员，与必须搜索表中的所有行相比，索引会帮助程序员更快地获得该信息。

由于在整个银行系统中明细表是操作最频繁的，下面就为交易信息表的卡号列创建索引，以提高查询速度，代码如示例代码 18-13 所示。

示例代码 18-13

```
--创建索引：给交易明细表的卡号 cardID 字段创建索引
if exists(select * from sysindexes where name='index_cardID')
drop index index cardID on transInfo
go
create   NONCLUSTERED   INDEX   index cardID   ON   transInfo(cardID)WITH
FILLFACTOR=10
go
--按指定索引查询丹尼尔（卡号为 4942 1234 5678）的交易记录
SELECT * FROM transInfo with(INDEX=index_cardID) WHERE cardID='4213 4942
1234 5678'
```

注意：索引能够提高查询速度，但是同时又降低了增删改的速度，降低了数据库维护的效率，占用了更多的空间，上述代码中 FILLFACTOR=10 的原因在于插入的频率远大于查询，所以多预留些空间。另外查询索引是否存在查询的是 sysindexes 表，删除索引的语句也有所区别，需要指定表名。

18.3.3　创建存储过程

在编程的过程中，为了团队开发和代码重用，最基本的就是使用方法把一系列操作封装起来，方便调用，在 SQL 中，这种思想同样适用，也可以把完成一系列操作的 SQL 语句组整合在一起形成一个“方法”，在使用的时候，只需要调用即可。这就是存储过程，使用存储过程还有一些优势，如减少网络流通量、安全性高等。下面，就针对存款、取款、转账等操作建立存储过程，完成转账业务。

分析：存取款实质上是执行 2 个新增语句，但是在存取款之前必须进行用户合法性验证、账户是否激活验证、余额是否足够验证，将用到大量的 T-SQL 编程知识点。

1. 存取钱存储过程

存钱不需要验证密码，但是需要验证卡是否挂失锁定，取钱需要验证密码和余额，具体流程如图 18-10。

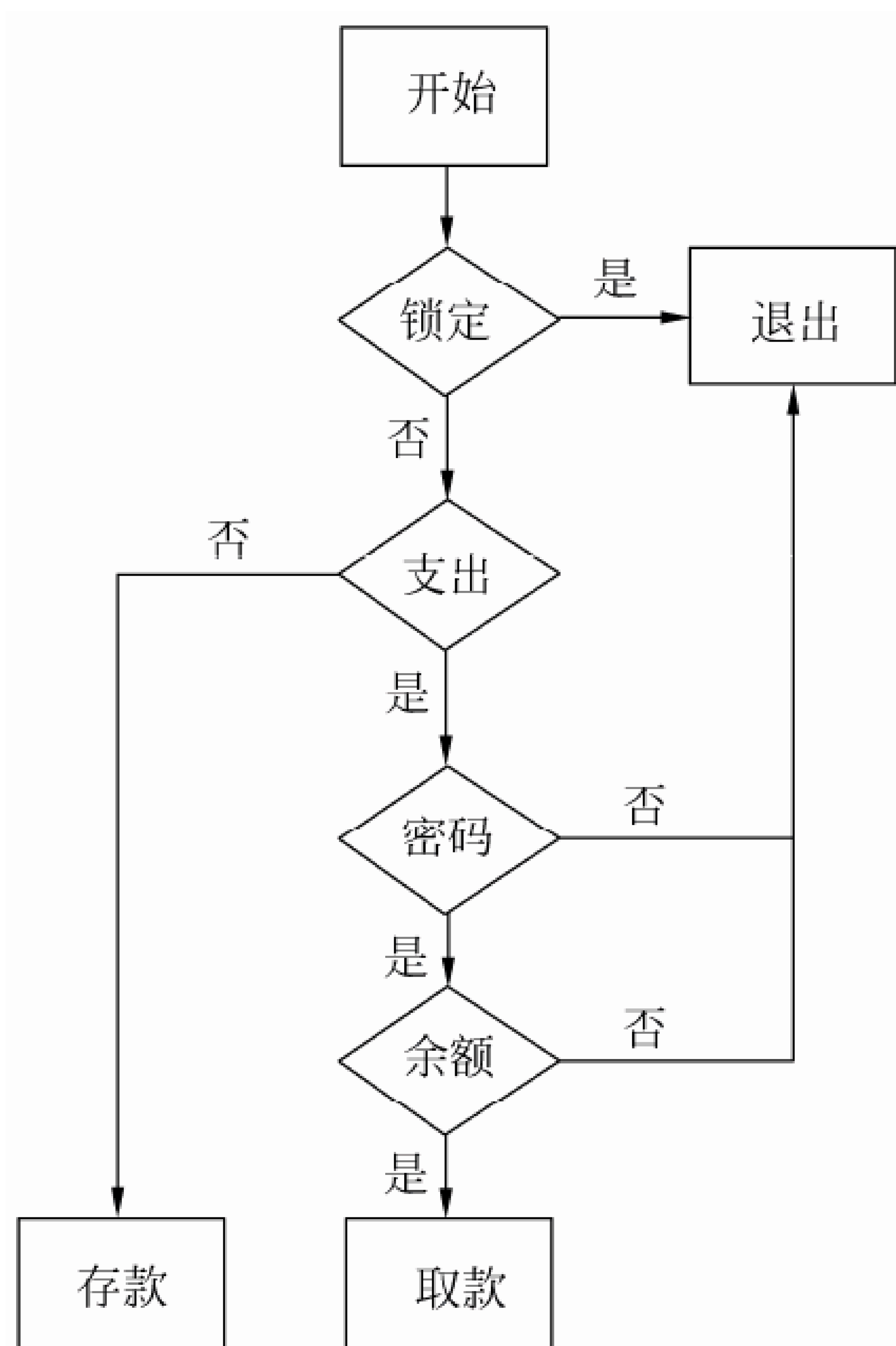


图 18-10　存取钱流程

用户请求使用本系统，系统需要做的第一件事情就是判定此卡是否锁定（挂失）。如果锁定则提示用户“此卡被锁定！”，在没有被锁定的情况下再判断是“支出”操作还是“存入”操作，存入操作不需要验证密码，直接进行操作。支出操作需要验证密码和余额，有一项不符合，系统就会提示“密码错误！”或“余额不足本次操作！”退出操作。最后根据业务更新或插入相关的表，所需要的参数为全部业务参数和，如卡号、金额、类型、密码，另外密码需要设置为带默认值的参数，因为存款不需要密码。全部存储过程创建代码如示例代码 18-14 所示。

示例代码 18-14

```
--取钱或存钱的存储过程
if exists(select * from sysobjects where name='proc_takeMoney')
  drop proc proc_takeMoney
go
create  procedure  proc_takeMoney  (@card  char(19),@money  money,@type
char(4),@passWord char(6)='')
 AS
    print '交易正在进行,请稍后......'
     if ((SELECT isLoss FROM cardInfo WHERE cardID=@card)=1)--检测卡是否被
     锁定
          begin
               raiserror ('卡已被锁定，请联系银行工作人员！',16,1)--打印错误信息
               return
          end
    if (@type='支出')
      if ((SELECT passWord FROM cardInfo WHERE cardID=@card)<>@passWord)--
      检测密码是否正确
         begin
           raiserror ('密码错误或者卡已被锁定！',16,1)
           return
```

```
        end
    DECLARE @balance money
    SELECT @balance=balance FROM cardInfo WHERE cardID=@card
    if (@type='支出')
       if (@balance>@money+1)--判断余额是否足够
           update cardInfo set balance=balance-@money WHERE cardID=@card
       else
          begin
            raiserror ('余额不足! 不能交易! ',16,1)
            print '卡号'+@card+'  余额: '+convert(varchar(20),@balance)
            return
          end
    else
        update cardInfo set balance=balance+@money WHERE cardID=@card
     INSERT INTO transInfo(transType,cardID,transMoney) VALUES(@type,
     @card,@money)--插入明细账
    print '交易成功! 交易金额: '+convert(varchar(20),@money)
    SELECT @balance=balance FROM cardInfo WHERE cardID=@card
    print '卡号'+@card+'  的余额是: '+convert(varchar(20),@balance)
GO
```

注意：为了安全创建存储过程，和创建数据库、表一样，都需要预先判断是否存在，判断存储过程查询的表和判断表一样，都查询的是 sysobjects 表。

另外在 T-SQL 编程中需要注意条件判断、循环和传统编程语言思路是一致的，在 Java 或 C#语言中 if 下面只有一条指令的时候，花括号是可以省略的。同样在 T-SQL 编程中，if 下面只有一条指令的时候也是可以省略的，但是 T-SQL 中没有花括号，替代它用的是 begin-end，也就是使用 begin 替代“{”，用 end 替代“}”，有“begin”必须有对应的 end。

存储过程创建好，需要通过调用并核对数据来测试业务流程正确与否，以华盛顿为例，测试锁定的情况，再以丹尼尔为例测试密码错误的情况和余额不足的情况，对比数据。代码如示例代码 18-15 所示。

示例代码 18-15

```
--测试存储过程取钱操作
exec proc_takeMoney '4213 4942 1212 1134',10,'支出','123456'--卡被锁定的
                                                              情况
exec proc_takeMoney '4213 4942 1234 5678',10,'支出','123456'--密码错误情况
exec proc_takeMoney '4213 4942 1234 5678',10,'支出','888888'--正常情况
exec proc_takeMoney '4213 4942 1234 5678',6990,'支出','888888'--余额不足的
                                                                情况
exec proc_takeMoney '4213 4942 1234 5678',50,'存入'             --存入情况
```

调用存储过程用 exec（是 execute 的缩写形式）关键字，后面参数依次按照声明存储过程时的参数顺序和类型排列，这是直接调用法。还有一种使用变量传递参数法，参数的位置可以随意互换，如，存储过程原型是

```
create proc proc_getPassStudent(@written int,@lab int)—根据笔试和机试查询考
试通过学生信息
```

可以这样调用

```
exec proc_getPassStudent 60,70
```

也可以这样调用：

```
exec proc_getPassStudent @written=60,@lab=70—或
exec proc_getPassStudent @lab=70,@written=60
```

上例用变量名传递之后的写法如下：

```
exec proc_takeMoney @card='4213 4942 1212 1134',@money=10,@type='支出',@passWord='123456'          --卡被锁定的情况
exec proc_takeMoney @type='支出',@money=10,@card='4213 4942 1212 1134',@passWord='123456'          --卡被锁定的情况
```

2. 产生卡号存储过程

目前现行的银行卡号是 16 位数字组成（每 4 位一组，中间用空格隔开，共计 19 位），按照国家的统一规划，央行统一发放给各个商业银行之间的业务号段。每个行的号段都不相同，每个地区每种银行的前 8 位是固定的，后面 8 位由银行进行分配。可以采取的分配手段有依次加 1 和随机产生，但是不管怎么做，号码都必须是唯一的。下面的主要任务就是把产生出后面 8 位数字和前面的 8 位业务号码进行组合，形成新的卡号。要产生一个随机数需要用到 MS SQL 2005 的随机函数 rand()。rand()的用法如 rand（种子）返回从 0～1 之间的随机 float 值，使用同一个种子值重复调用 rand()会返回相同的结果。为了使产生的数字不同，就需要及时改变随机数的种子，以时间的毫秒部分为例，取随机数代码如：

```
select @r=rand(datepart(ms,getdate())*1000)—使用了 MS SQL 2005 的时间函数
```

由于毫秒不停地变化，所以产生出的结果也不一样，基本可以满足要求。下面就按照业务要求创建一个带输出参数的存储过程来实现卡号的产生功能。具体思路如：调用随机函数产生一个小于 1 的浮点数字，强制转换成为长度为 10 的字符串，截取 0.XXXXXXXX 的小数点后 1～4 位和 5～8 位组合为规定格式的卡号，然后使用输出参数传递出来，具体代码如示例代码 18-16 所示。

示例代码 18-16

```
if exists(select * from sysobjects where name='proc_randCardNo')
  drop proc proc randCardNo
go
--使用输出参数传递生成的卡号
if exists(select * from sysobjects where name='proc_randCardNo')
  drop proc proc_randCardNo
go
create proc proc_randCardNo(@randCardID nvarchar(19) output)
as
declare @str char(10)
declare @r numeric(10,8)
--在单个查询中反复调用 RAND() 将产生相同的值，所以使用 rand([seed])时，每次要修改种子 seed 的才能得出不一样的随机数
--这里采用毫秒做种子来取随机数
select @r=rand(datepart(ms,getdate())*1000)
select @str=convert(char(10),@r)--转化为 char 类型
```

```
--截取.XXXXXXXX 的小数点后-4 位和-8 组合为规定格式的卡号
set @randCardID='4213 4942 '+SUBSTRING(@str,3,4)+' '+SUBSTRING(@str,7,4)
go
```

这里用到存储过程的输出参数，声明输出参数只需要在类型后面加上 OUTPUT 关键字，在存储过程内部给这个参数赋值即可。但是接收就比较麻烦了，必须在外面声明一个同样类型的变量来接收，同时调用时也需加上 OUTPUT 关键字。产生卡号存储过程调用如下

```
--测试生成卡号存储过程，必须声明一个变量接收输出的值
declare @randCardID nvarchar(19) --
exec proc_randCardNo @randCardID output
print @randCardID--打印产生的随机卡号
```

注意：这个存储过程用到输出参数和类型转换函数及字符串截取函数，具体使用方法可以参考前面的章节和 Microsoft SQL Server 2008 联机丛书。学习项目不仅是学习如何实现，更重要的在于学习为什么要这样实现，怎么用技术解决现实问题，遇到问题之后怎么办。

3. 开户存储过程

上面的存储过程虽然已经产生出了卡号，但是开户还需要一些其他的操作，比如插入客户的基本信息，包括用户名、身份证号、电话地址等，为了简化编程操作，把生成卡号操作封装成一个存储过程，下面的开户只需要调用即可。同时经过详细的分析大家会发现，虽然使用了毫秒作为随机数的种子，但是重复的几率比较大。银行卡号涉及财产安全，要求的是绝对不能重复，这个问题也需要在开户操作中完成。思路如：把开户需要的参数全部传入，先调用产生卡号的存储过程产生一个卡号，然后在卡信息表中查询，如果已经存在该号码则继续产生，直到不存在为止。要解决这个问题就需要用到 SQL 的另一种语法：while 循环。while 循环的思路如图 18-11 所示。

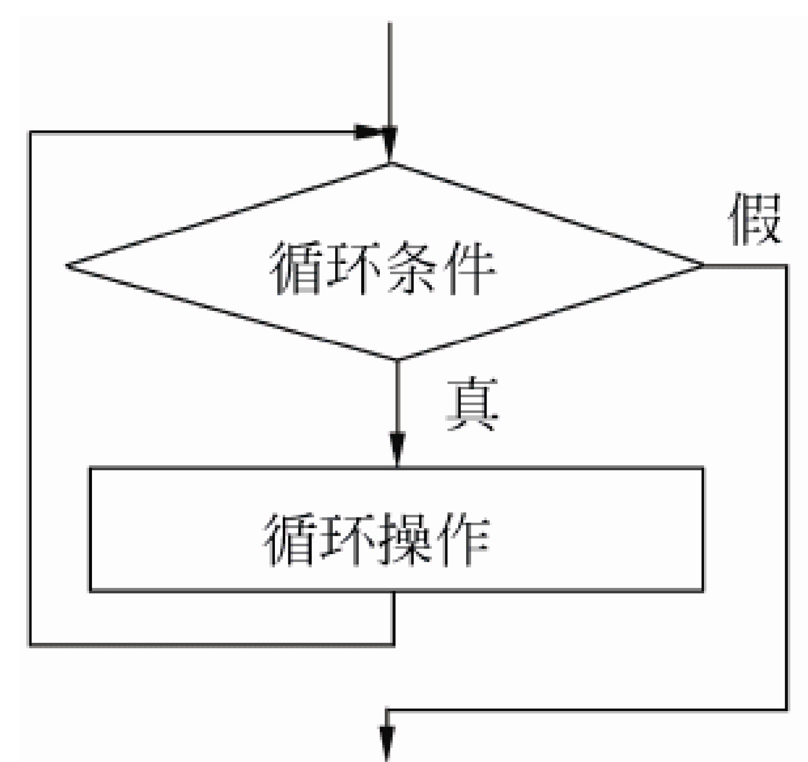


图 18-11　循环流程

按照这个思路，把卡号是否重复作为循环条件，把产生卡号作为循环操作就能产生出一个绝对不重复的卡号。编写出的存储过程如示例代码 18-17 所示。

示例代码 18-17

```
if exists(select * from sysobjects where name='proc_createAccount')
  drop proc proc createAccount
go
create  procedure  proc_createAccount  @customerName  char(8),@customerPID
char(18),@telephone varchar(13)
    ,@openMoney money,@savingType char(8),@address varchar(50)=''
  as
    DECLARE @mycardID char(19),@cur_customerID int
    --调用产生随机卡号的存储过程，生成随机卡号
    EXEC proc randCardNo @mycardID OUTPUT
    while  exists(SELECT * FROM cardInfo WHERE cardID=@mycardID) --如果生
```

```
    成的卡号有重复，就一直产生，直到没有重复为止
      EXEC proc randCardNo @mycardID OUTPUT
    print '尊敬的客户,开户成功!系统为您产生的随机银行卡号为:'+@mycardID
    print '开户日期'+convert(char(10),getdate(),111)+'  开户金
    额:'+convert(varchar(20),@openMoney)
    IF not exists(select * from userInfo where customerPID=@customerPID)
       --插入客户信息表
     insert into userInfo(customerName,customerPID,telephone,address )
       values(@customerName,@customerPID,@telephone,@address)
       --查询出最后一次插入的自增列值
    select @cur customerID=@@identity from userInfo
       --插入卡信息表
    insert into cardInfo(cardID,savingType,openMoney,balance,customerID)

values(@mycardID,@savingType,@openMoney,@openMoney,@cur_customerID)
       --插入明细账
     insert into transInfo(transType,cardID,transMoney) values('存入',
     @mycardID,@openMoney)
GO
```

这个存储过程还用到了查询最后一次自增列的值@@identity 全局变量，@@identity 是系统变量，可以直接调用，在插入操作执行完之后它存储的是自增列的值。由于上述存储过程在插入客户信息表之后还要插入卡信息表，而卡信息表和客户信息表是有主外键关系的，所以需要获取新产生的客户 ID，最后要把开户操作记录明细账。测试此存储过程的代码如示例代码 18-18 所示。

示例代码 18-18

```
--测试开户
exec proc_createAccount '张三','123456789387654321','029-88888888',1200,'
定期','西安'
--查看插入结果
select * from userInfo
select * from cardInfo
select * from transInfo
```

4．转账业务（带事务）

ATM 交易管理系统最重要的功能之一莫过于转账操作，转账基本业务是把客户 A 的相应金额转到客户 B 账户。整个过程最少进行 2 次数据库操作，要求不能有任何错误出现，要么全部 SQL 语句执行成功，要么全部失败，一旦出现任何不完整的操作，后果都是不堪设想的，所以要采用事务处理这个业务。事务的本质就是保证操作的完整性和一致性，具有原子性的特征，意思是无论多少操作都被看做是一个整体，要么全部成功，要么全部失败。使用事务一般只需要知道 3 句话即可，开启事务：begin tran，提交的基本用法举例如示例代码 18-19 所示：

示例代码 18-19

```
begin tran
    declare @err int
    set @err=0
    upate ……
```

```
    set @err=@err+@@error—累加错误号
    insert into ……
    set @err=@err+@@error
    ...
if @err<>0
    rollback tran   --回滚事务，相当于什么都没发生
else
    commit tran     --提交事务，提交更改，永久保存
```

转账操作也需要验证用户的合法性和卡号是否锁定、余额是否足够等，需要的参数有转出卡号、转入卡号、金额和密码，验证通过后需要记录明细账，并根据实际情况提示操作结果，代码如示例代码 18-20 所示。

示例代码 18-20

```
--转账事务
if exists(select * from sysobjects where name='proc_tranMoney')
  drop proc proc_tranMoney
go
create  proc  proc tranMoney(@fromCard  char(19),@toCard  char(19),@money
money,@pass char(6))
as
    print '正在转账,请稍后......'
    declare @err int,@blance money--声明
    set @err=0
     --验证用户合法性
     if exists(select * from cardInfo where cardID=@fromCard and
     passWord=@pass)
       begin
         select @blance=balance from cardInfo where cardID=@fromCard
         if(@blance>@money+1)
             --可以支取
             begin
                 begin tran
                 update cardInfo set balance=balance-@money WHERE cardID=
                 @fromCard
                 set @err=@err+@@error
                 insert into transInfo(transType,cardID,transMoney) values
                 ('支出',@fromCard,@money)--插入明细账
                 set @err=@err+@@error
                 update cardInfo set balance=balance+@money WHERE cardID=
                 @toCard
                 set @err=@err+@@error
                 insert into transInfo(transType,cardID,transMoney) values
                 ('存入',@toCard,@money)--插入明细账
                 set @err=@err+@@error
                 if (@err>0)
                     begin
                     print '转账失败，请联系工作人员！'
                     rollback tran
             end
    else
      begin
        print '转账成功！'
        commit tran
      end
             end
         else
```

```
            print '余额不足本次交易，交易取消！'
       end
     else
         print '密码错误，交易取消！'
go
```

现模拟从丹尼尔账户转账 40 元到张三账户，并显示转账结果。同时测试其他的情况，如密码错误、余额不足等。测试代码如下：

```
--测试转账-正常情况
proc_tranMoney '4213 4942 1234 5678','4213 4942 9804 3224',40,'888888'
--测试转账-密码错误
proc tranMoney '4213 4942 1234 5678','4213 4942 9804 3224',40,'888887'
--测试转账-余额不足
proc_tranMoney '4213 4942 1234 5678','4213 4942 9804 3224',7000,'888888'
```

由于带事务，所以上述操作的完整性会得到保障。在类似于一个业务需要多个操作的情况下都需要加上事务处理，才能保证数据准确一致。

18.3.4　创建数据库账户

所有数据库相关业务操作创建完毕，在正式投入使用之前必须保证数据库的安全。一般开发使用的都是数据库管理员，权限最高，但是这样的账户一旦被暴露出去，将会取得数据库所有操作权限，甚至于删除数据库。因此保障数据的安全就成为这个阶段的头等大事，所以，在给客户部署的时候。特别是一个数据库服务器上有多个应用数据库的情况下，需要根据实际情况分配不同的权限，限制其操作。

分配权限从创建登录账户开始，分为 3 步走：登录数据库服务器、访问数据库、操作表。

1. 创建登录账户

数据库的登录账户有 2 种，一种是 SQL 登录账户，只能登录 Microsoft SQL Server 服务器，还有一种是 Windows 登录账户，Microsoft® Windows NT 用户或组账户得以使用 Windows 身份验证连接到 Microsoft SQL Server，一般不常用。SQL 登录创建语法如示例代码 18-21 所示。

示例代码 18-21

```
--添加 SQL 登录账号
If not exists(select * from master.dbo.syslogins where loginname='admin')
   begin
     exec sp_addlogin 'admin', '1234'    --添加 SQL 登录账号
   end
  go
```

2. 访问数据库

创建登录账户之后，说明该用户可以使用分配的密码登录数据库服务器了，但不能对数据库进行操作。接下来就再次给该用户分配数据库权限，使用的脚本如示例代码 18-22

所示。

示例代码 18-22

```
--创建数据库用户
 use  BankMIS
 go
 exec sp grantdbaccess  'admin', 'DBUser'
 go
```

🔔注意：执行 sp_drantdbaccess 存储过程时，需要两个参数，第一个是登录名，第二个是数据库账户名，两个参数可以一样也可以不一样，一般建议设置成一样的，方便管理。另外，这句话需要指定授权的数据库。

3. 访问表

这样就可以登录到指定的数据库，但是看不见任何用户表，也不能操作，接下来还得对表进行授权，对表授权就比较详细了，可以分别指定用户具备增、删、改、查 4 种操作的任意一种或者多种组合，只有授过权的操作才会执行，没有权限的操作会被拒绝。代码如示例代码 18-23 所示。

示例代码 18-23

```
--给数据库用户授权
 --为 DBUser 分配对象权限（增删改查的权限）
 grant select  on transInfo to DBUser --对于表 transInfo，用户 DBUser 只能
 查询
 select * from  transInfo            --允许
 delete from transInfo               --拒绝
 grant all  on userInfo to DBUser    --对用户 DBUser 授予全部操作
 grant select,insert,update,delete,select  on cardInfo to DBUser
                                     --对用户 DBUser 授予增删改查操作权限
GO
```

🔔注意：上述权限控制语句的测试需要在使用创建的用户登录之后执行。如果在当前工作环境执行，用的权限则是你当前登录的账户，看不到权限控制的效果。

18.4　项 目 小 结

通过本章 ATM 交易管理系统整个开发流程的学习，读者对数据库设计分析及创建和常用 SQL 编程应该有一个比较清晰的认识。数据库编程有时能在很大程度上降低程序开发成本，所以掌握数据库开发是非常必要的，特别是存储过程和一些常见函数。在开发中经常会用到，希望读者能从本次的项目中有所收获，能够在以后的开发中活学活用。如果读者还对界面开发感兴趣，可以通过本例的数据库代码，自己设计界面实现一个真正的 ATM 机操作系统。

第 19 章　ADO.NET 实例——进销存管理信息系统

目前，在我国使用计算机软件进行企业信息化管理已经成为了一种社会发展的必然趋势，凡是日常生活中需要进行手工操作的工作，都可以使用计算机软件来代替。当然作为商业企业，都需要对商品进行采购、销售和日常管理。随着企业业务不断地增多，商品种类和数量的成倍增长，客户量的增大，导致企业日常的管理和运作需要更多的人手。鉴于此，若想要提高劳动效率和管理水平，降低运营成本，必然需要借助计算机软件进行企业信息化管理。本系统正是在这种商业企业需求下应运而生的。

19.1　项 目 概 述

因某企业商品管理需要，本软件公司按照其要求开发一套进销存管理系统，提供便捷、准确和可靠的操作，以便方便对商品的信息管理。该系统的操作界面简洁、实用，软件帮助系统图文并茂，让使用者可以在最短的时间内掌握软件的使用方法。

19.1.1　功能概述

通过初步的和客户进行沟通，确认这个系统需要实现的主要功能包含以下部分。

- 用户登录验证。
- 商品信息维护。
- 用户信息维护。
- 采购入库。
- 采购退货。
- 销售出库。
- 销售退货。
- 库存查询。
- 订单查询。

19.1.2　用户环境描述

软件系统采用分布式架构，整个项目需要一个主服务器，每个商品管理分部都需要有至少一台终端，24 小时不间断服务。

19.1.3　可行性分析

从投资角度分析，本项目能够提高业务的运转效率和企业市场竞争力，采用计算机软件管理可以降低运营成本，以便让企业获取更大的利润。

从技术可行性角度分析，目前.NET 平台已经非常成熟，并且 SQL Server 2008 已经成为各个大中型企业优先采用的数据库服务器，完全可以满足本企业需要。

从社会可行性角度分析，采用软件管理业务已经成为一种绝对趋势，是先进管理思想的一种体现，并且操作人员为了减少手工记账的错误也乐意接受使用计算机软件进行管理的方式。

该系统运行在 Windows 平台上，需要有.NET 运行环境和 SQL Server 2008 数据库服务器。采用镜像技术可以提高整个系统的运行效率，以更高效率地进行业务处理，给企业带来更高的利润。

19.2　项目需求分析

需求分析是对用户的业务活动进行分析，确定系统的目的、范围、定义和功能，明确在用户的业务环境中软件系统应该“做什么”。只有在确定了客户需求后，知道要“做什么”，才能够分析和寻求系统的解决方法，开展后续的工作，所以需求分析是软件工程中的一个关键过程。

19.2.1　系统功能性需求分析

在进销存管理软件的需求采集过程中，首先，要明确谁才是真正的软件使用者，因为只有软件使用者才能真正知道需要什么样的软件。进销存管理软件的使用者，主要为系统管理人员、采购人员和销售人员，而且他们对软件有着不同的需求。可以通过以下 3 个方面收集需求。

（1）收集商品销售人员所希望的商品管理系统需求。

由于该软件的一部分用户是商品销售人员，所以需要有选择地与一些具有代表性的商品销售员进行面对面交流，咨询他们对本软件的一些看法和观点。比如，您认为该软件应具备哪些功能等。重点是商品销售员需要具有代表性，至少每个商品销售部门都需要有 1～2 位销售人员接受调查。

（2）收集商品管理人员所希望的商品管理系统需求。

从商品管理人员的角度去了解软件需求。首先，有针对性地对部分商品管理员代表进行咨询，得到商品管理人员眼中该软件的一个大致框架和功能，重点在于商品信息管理部分。然后，再对广大商品管理人员进行广泛的问卷调查，提出更多更具体更细节的功能点，让商品管理人员选择。

（3）收集商品采购人员所希望的商品管理系统需求。

前面是从商品销售和管理人员的角度出发收集需求，接下来要从商品采购人员的角度

去了解软件需求。首先，有针对性地对部分商品采购员代表进行咨询，得到商品采购人员眼中该软件应该具备的功能，重点在于商品采购部分。然后，再对广大商品采购人员进行广泛的问卷调查，提出更多更具体更细节的功能点，让商品采购人员选择。

通过和客户的反复沟通及确认，最终总的业务流程如图 19-1 所示。

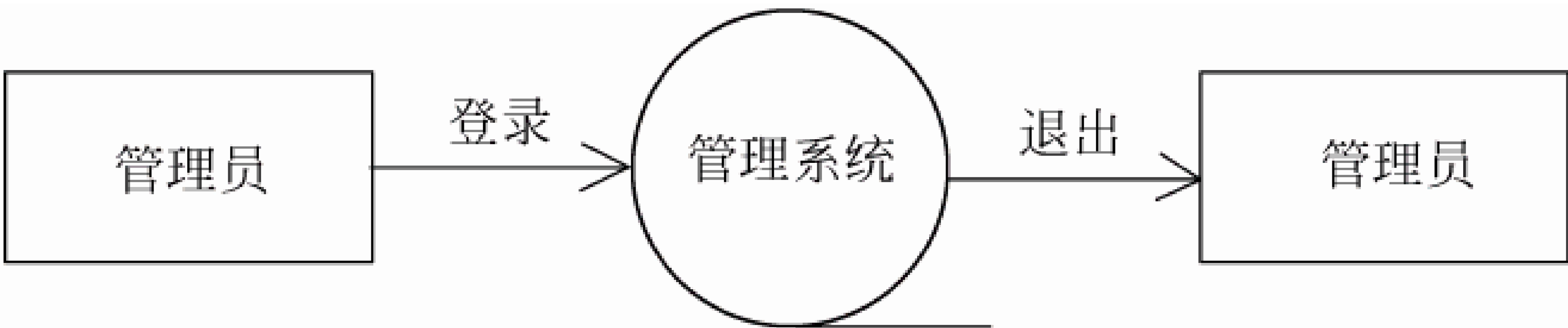


图 19-1　总流程图

根据客户描述的软件需求，从功能的角度进行初步的模块划分。通过对客户所描述的软件需求的理解，区分每个功能的边界，这是再一次细化的过程，分析结果如表 19-1。

表 19-1　进销存管理软件初步需求分析

功能＼属性	功　能　点	功 能 描 述	重　要　性
销售人员的需求	登录系统	输入账号和密码进行验证	重要
	查看商品信息	查看所有商品及某个商品的详细信息	重要
	销售出库	修改用户所选商品的数量（减少库存量）	重要
	销售退货	修改用户所要退商品的数量（增加库存量）	重要
采购人员的需求	登录系统	输入账号和密码进行验证	重要
	商品入库	增加新商品或修改已有商品数量	重要
	商品退货	删除商品或减少已有商品数量	重要
管理人员的需求	登录系统	输入账号和密码进行验证	重要
	查看商品信息	查看所有商品及某个商品的详细信息	重要
	修改和确认商品信息	对库存中的商品信息进行必要的修改、确认	重要

根据表 19-1 用户描述的需求，将系统模块划分如表 19-2 所示，并对其模块的划分和功能进行描述。划分模块的目的是使系统结构更加清晰，开发难度降低并且有利于团队开发。

表 19-2　进销存管理软件需求分析结果

功能＼属性	功　能　点	功 能 描 述	重要性
系统登录	登录流程	对用户输入的用户名、密码进行验证，验证通过后，该用户可以使用PSS系统中自己拥有权限的那部分功能，否则拒绝使用	重要
维护基本资料	商品资料维护	用户修改、删除、新增或查询商品数据，系统根据用户的操作，对商品资料进行更新或显示	重要
	供应商资料维护	用户修改、删除、新增或查询供应商数据（其中包括对供应商的联系人的修改，删除，新增与查询以及对供应商交易记录的查询），系统根据用户的操作，对供应商资料进行更新或显示	重要
采购商品	采购入库	用户通过录入采购入库单增加采购的货物，并可对采购入库单及其单据中的货物明细进行修改、删除与查询	重要

续表

属性 功能	功　能　点	功 能 描 述	重要性
	采购退货	用户录入通过采购退货单退回货物，并可对采购退货单及其单据中的货物明细进行修改、删除与查询	重要
销售商品	销售出库	用户通过录入销售出库单记录销售的货物，并可对销售出库单及其单据中的货物明细进行修改、删除与查询	重要
	销售退货	用户通过录入销售退货单退回已销售货物，并可对销售退货单及其单据中的货物明细进行修改、删除与查询	重要
库存管理	库存查询	用户通过组合不同查询条件，对库存商品进行查看、盘点	重要

认真比较表 19-1 和表 19-2 可以看出，表 19-1 是从用户的角度描述软件的需求，而表 19-2 是从开发人员的角度描述软件的需求，这体现了需求分析的主要目的——让开发人员懂得软件需要做什么。

19.2.2　系统总用例分析

在分析清楚系统功能需求之后，重要的就是把系统功能划分成为相对独立的模块或者子系统，便于集中讨论和确定需求，并且需要确定出操作用户，下面就根据功能用例图的方式直观的表现出该系统的业务流程，如图 19-2 所示。

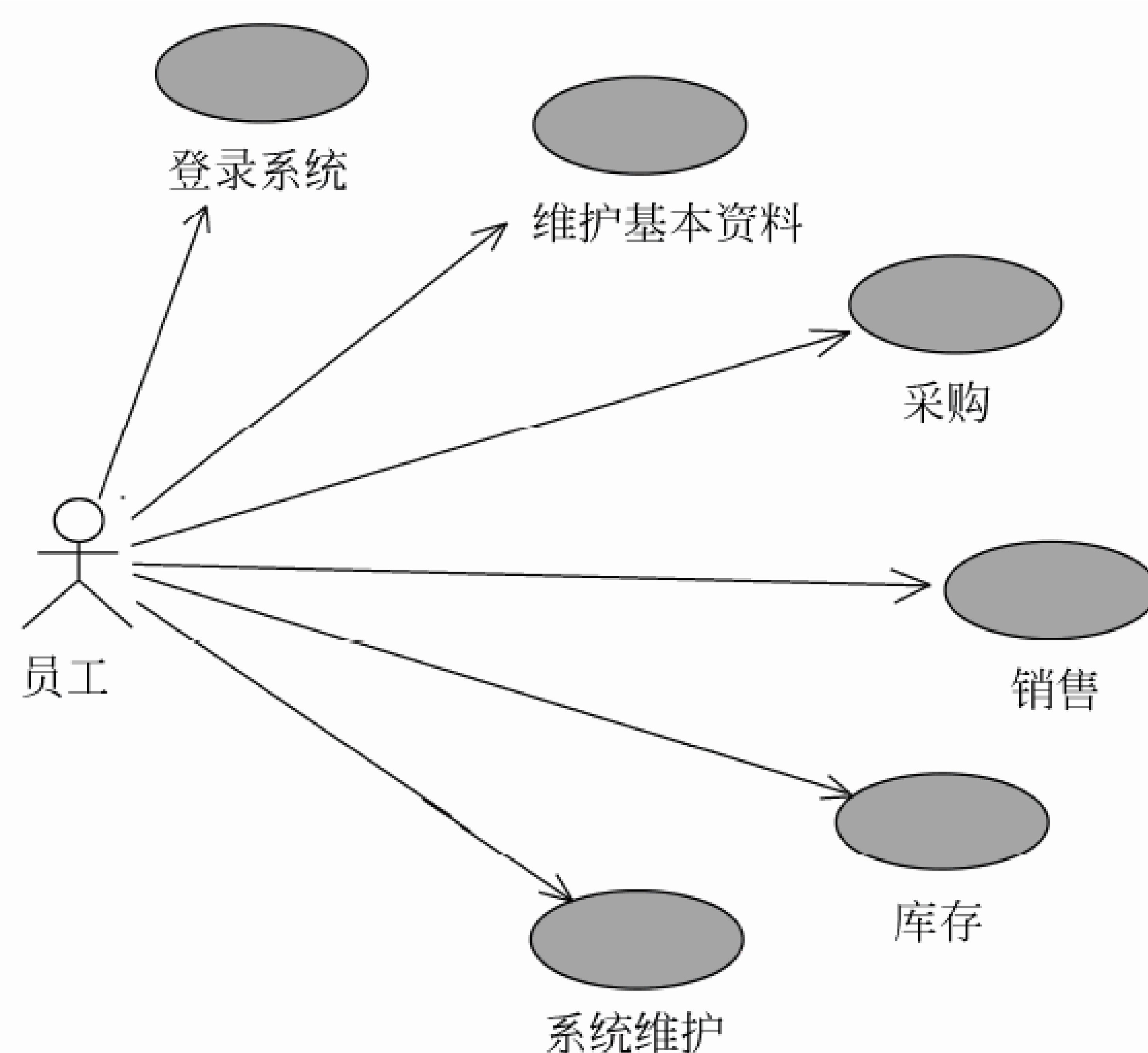


图 19-2　系统总用例

19.2.3　系统用例分析

【用例 1：基本资料维护】

- ❑ 描述：提供维护员工信息、维护商品信息、维护供应商信息和订单信息（注意删除操作，商品库存量不为 0 时不能执行删除商品操作）。
- ❑ 参与者：管理员。

- 用例图：图 19-3。

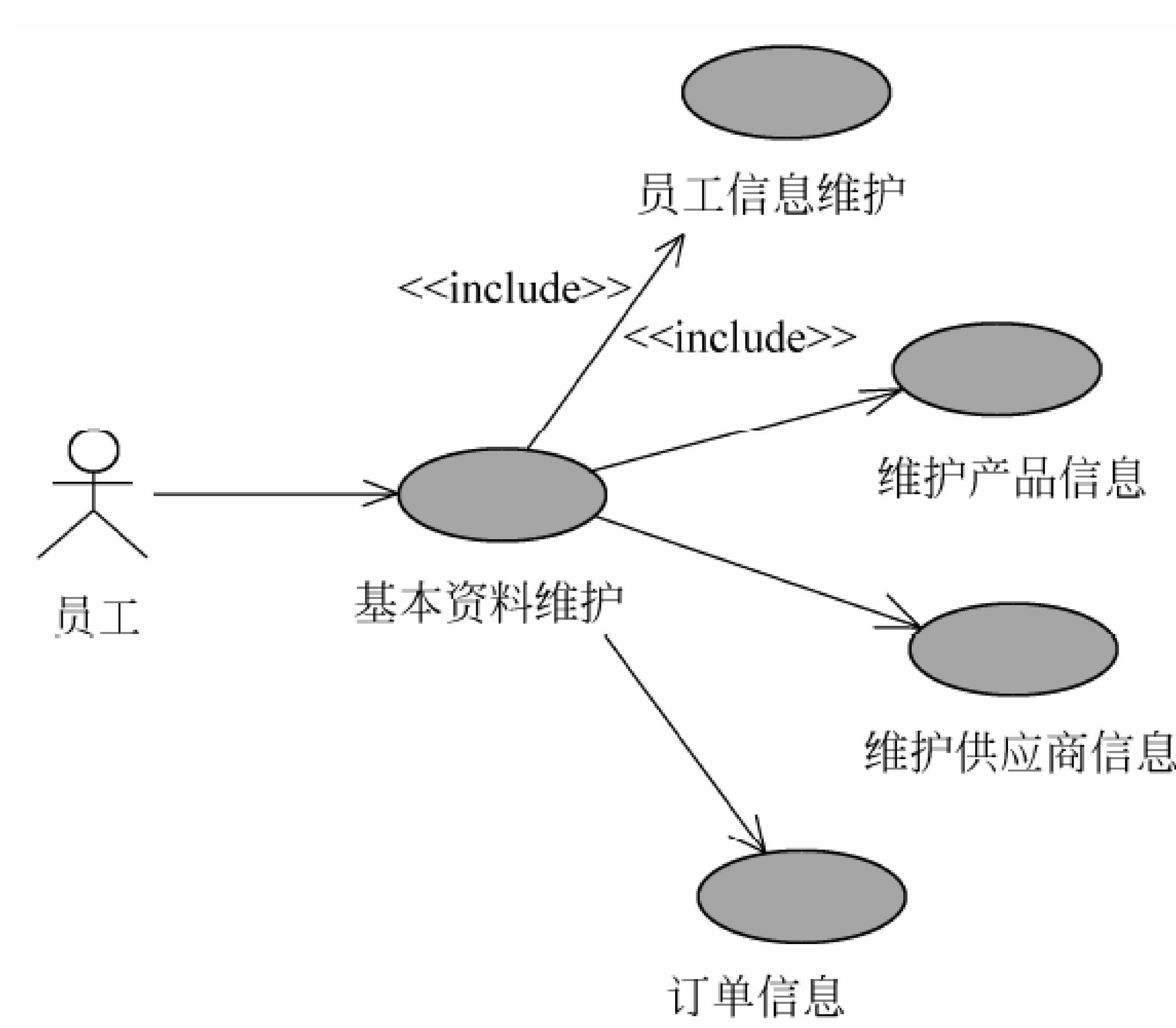


图 19-3　基本资料维护用例图

【用例 2：采购商品】

- 描述：提供采购商品入库（注意添加新商品操作，刚添加的商品库存量为 0 需要在采购界面输入采购数量）功能。
- 参与者：采购员。
- 用例图：图 19-4。

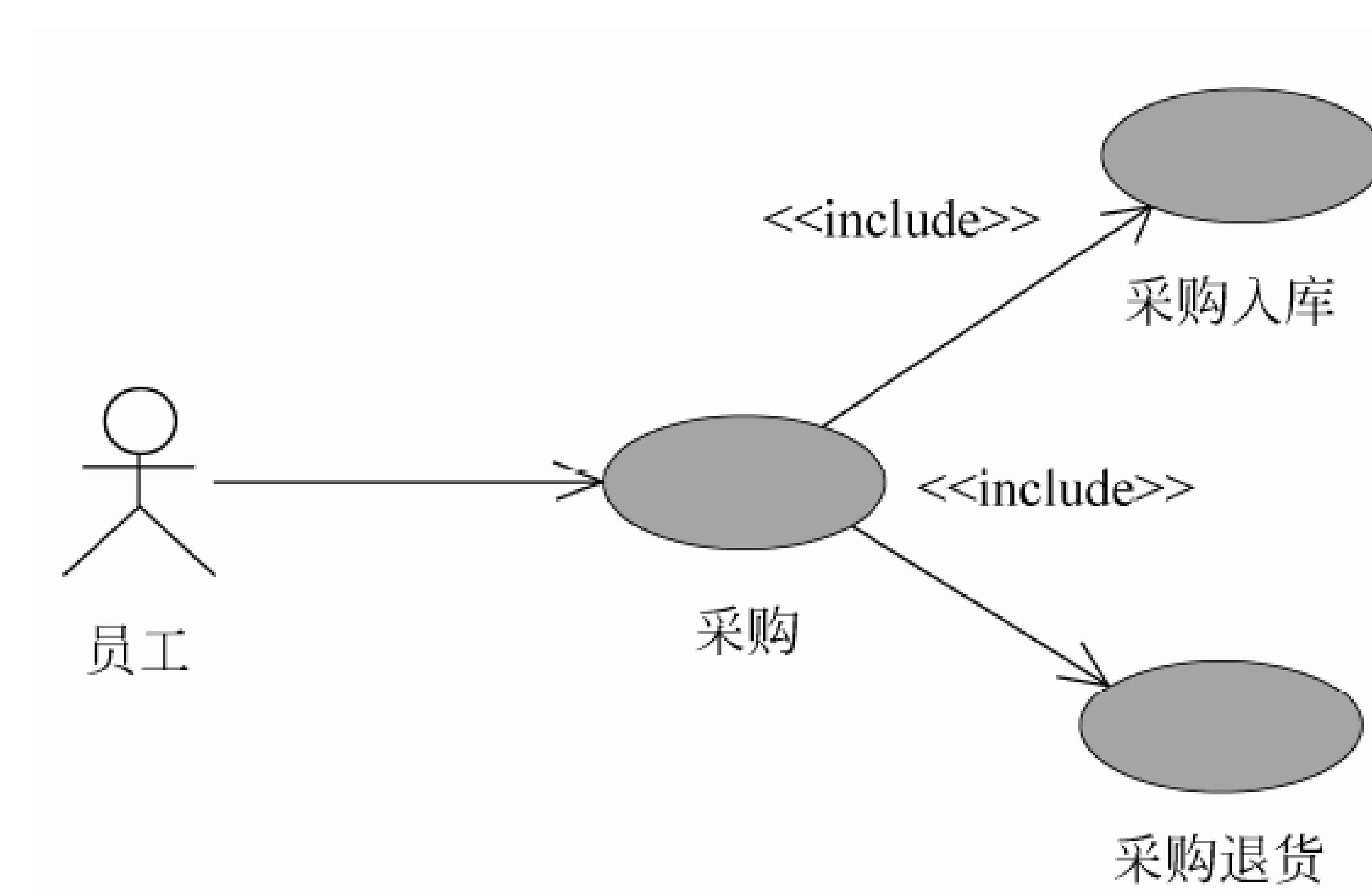


图 19-4　采购用例图

【用例 3：库存管理】

- 描述：提供盘点、查看库存商品信息及确认商品库存下限。
- 参与者：系统管理员、采购员、销售员。
- 用例图：图 19-5。

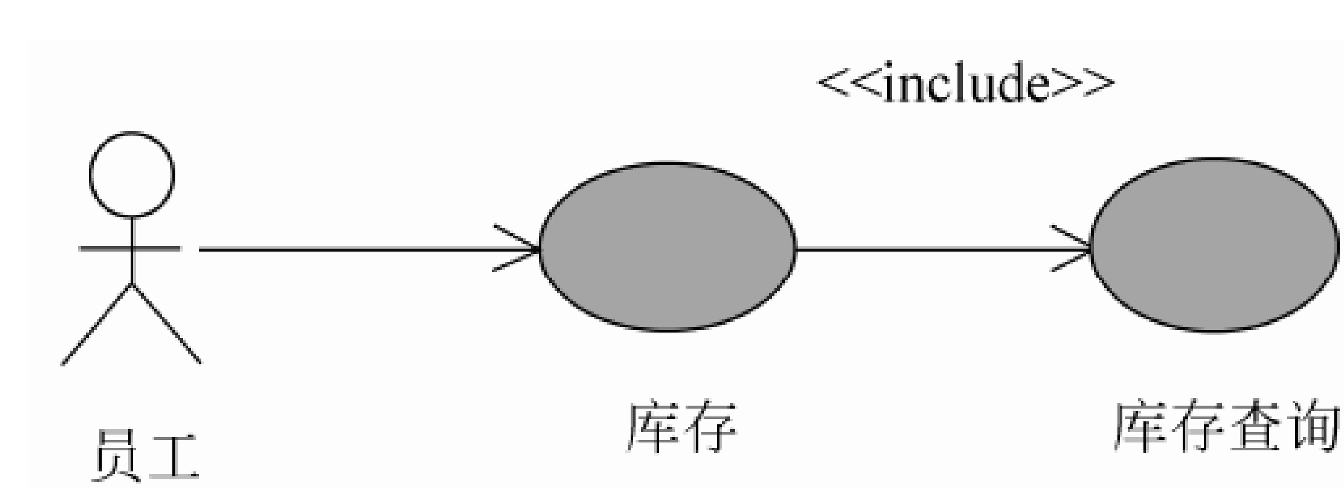


图 19-5　库存管理用例图

【用例 4：销售商品】

- 描述：修改商品库存量（注意商品销售操作，销售数量不能大于库存数量）。
- 参与者：销售员。
- 用例图：图 19-6。

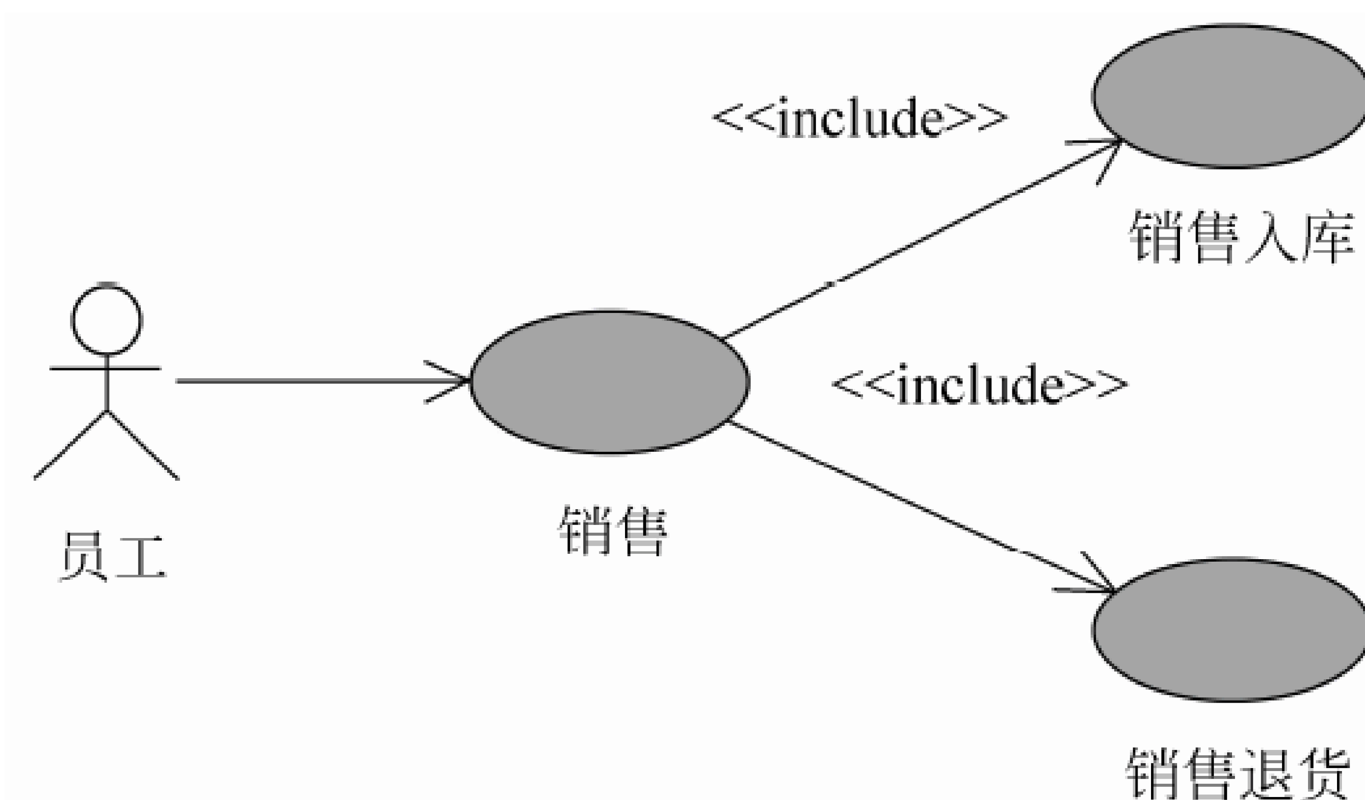


图 19-6　销售用例图

【用例 5：系统维护】

- 描述：对操作人员的权限限制、重新分配和资料维护。
- 参与者：管理员。
- 用例图：图 19-7。

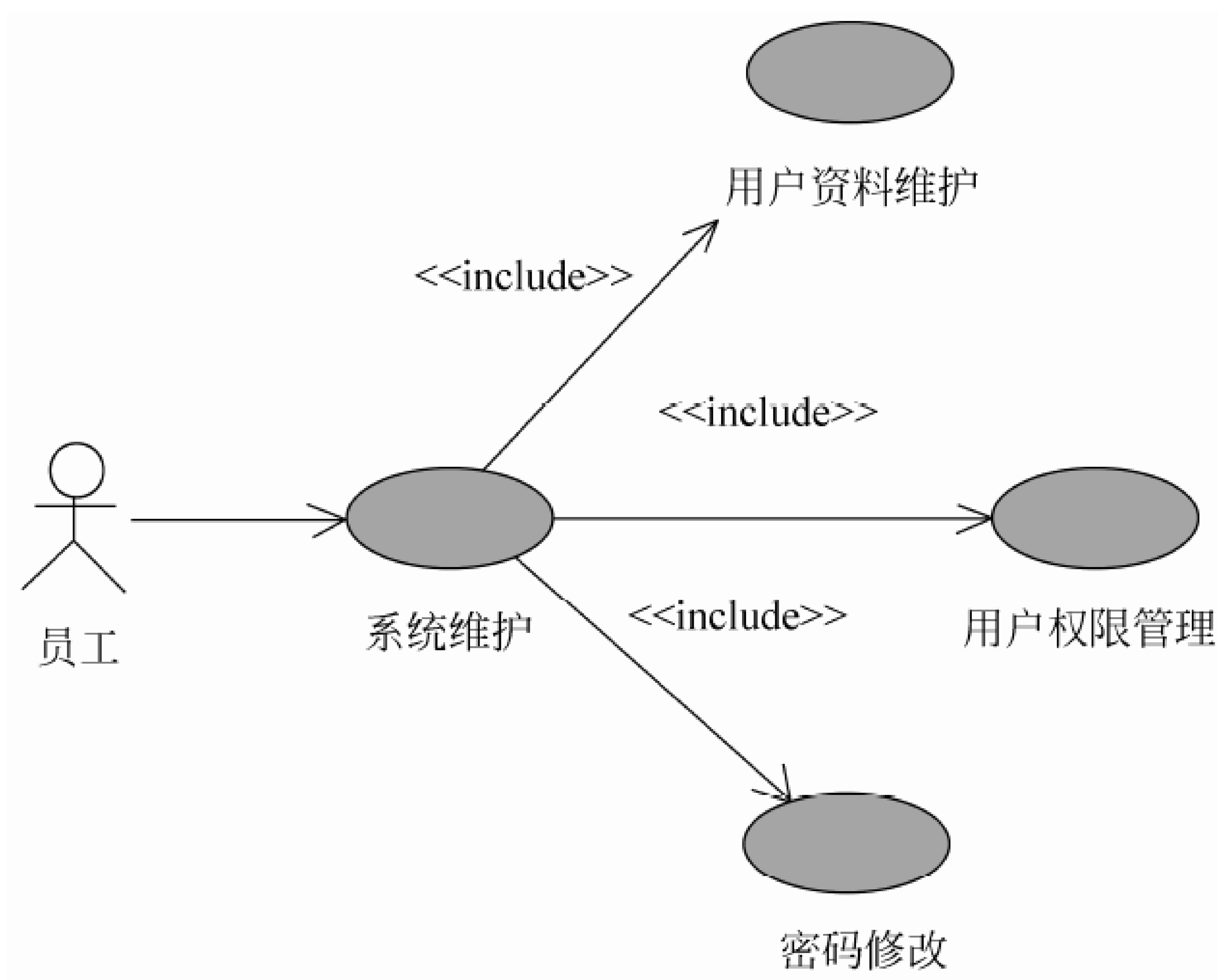


图 19-7　系统维护用例图

19.2.4　系统流程分析

在了解用户的需求之后，就需要确定系统的流程。简单一些的流程使用语言可以描述清楚，但是稍复杂一些的系统流程用文字描述就显得不够清晰了，这就需要使用流程图。进销存系统的整体流程图如图 19-8 所示。

注意：每一个小模块下面的文字表示可以进入该流程的用户类型。

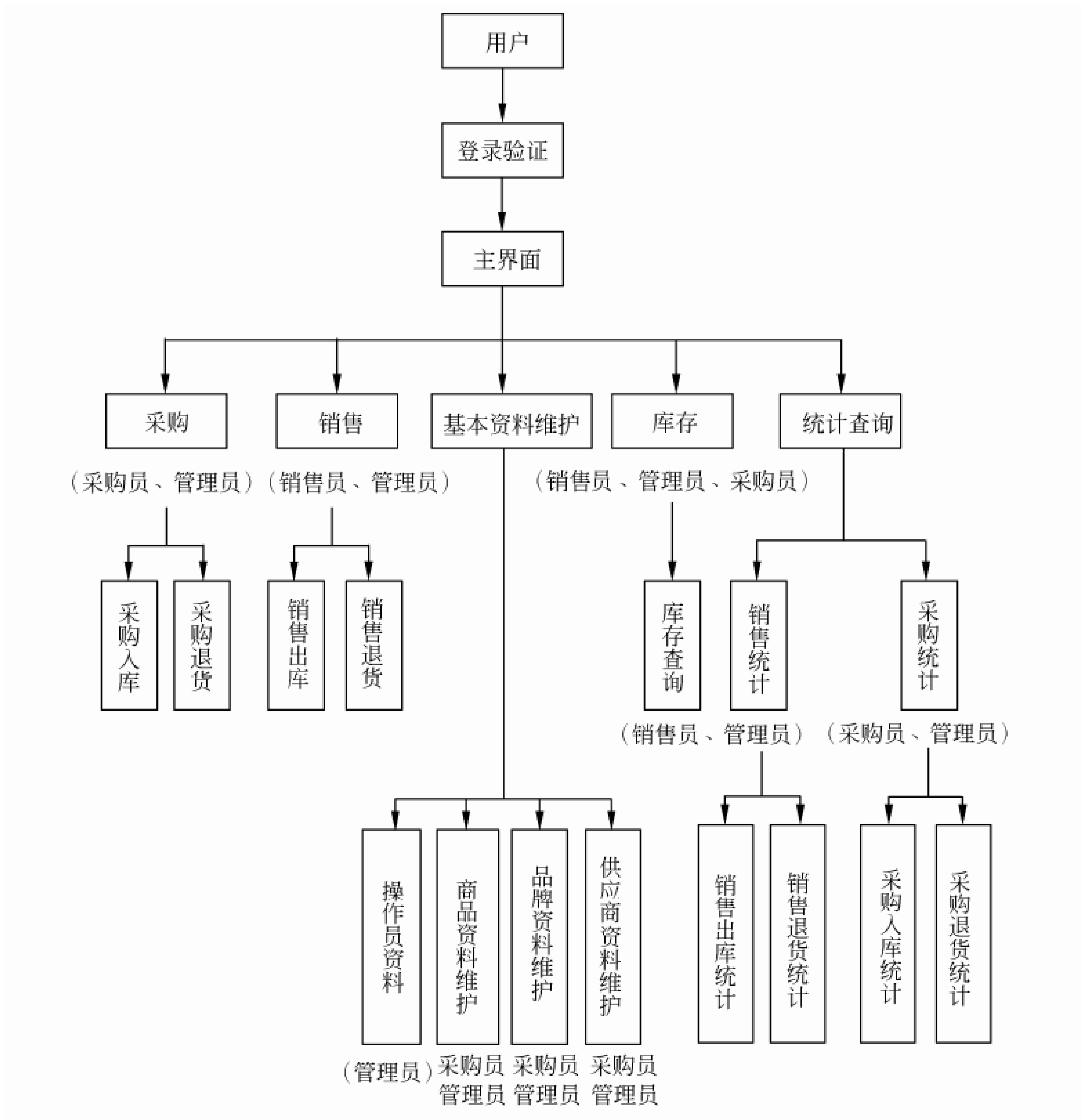


图 19-8　进销存系统流程图

19.2.5　模块分析

模块分析是描述系统需求的一个过程，需要将需求分析中的感性描述进行抽象，提取出要实现的功能，这是整个系统开发的一个关键过程。分析的根本目的是在开发者和提出需求的人之间建立一种理解和沟通的机制。因此，这个进销存系统的需求分析也应该由开发人员和用户或者客户一起完成。由于这里的例子只用于帮助读者学习系统开发的过程及方法，所以对于将要开发实现的进销存系统实际上并没有真正的用户或客户，在开发过程中假设笔者就是系统的使用者，并由此提出具体需求。

通过上面的信息收集和分析，最终确认系统整体的模块结构如图 19-9 所示。

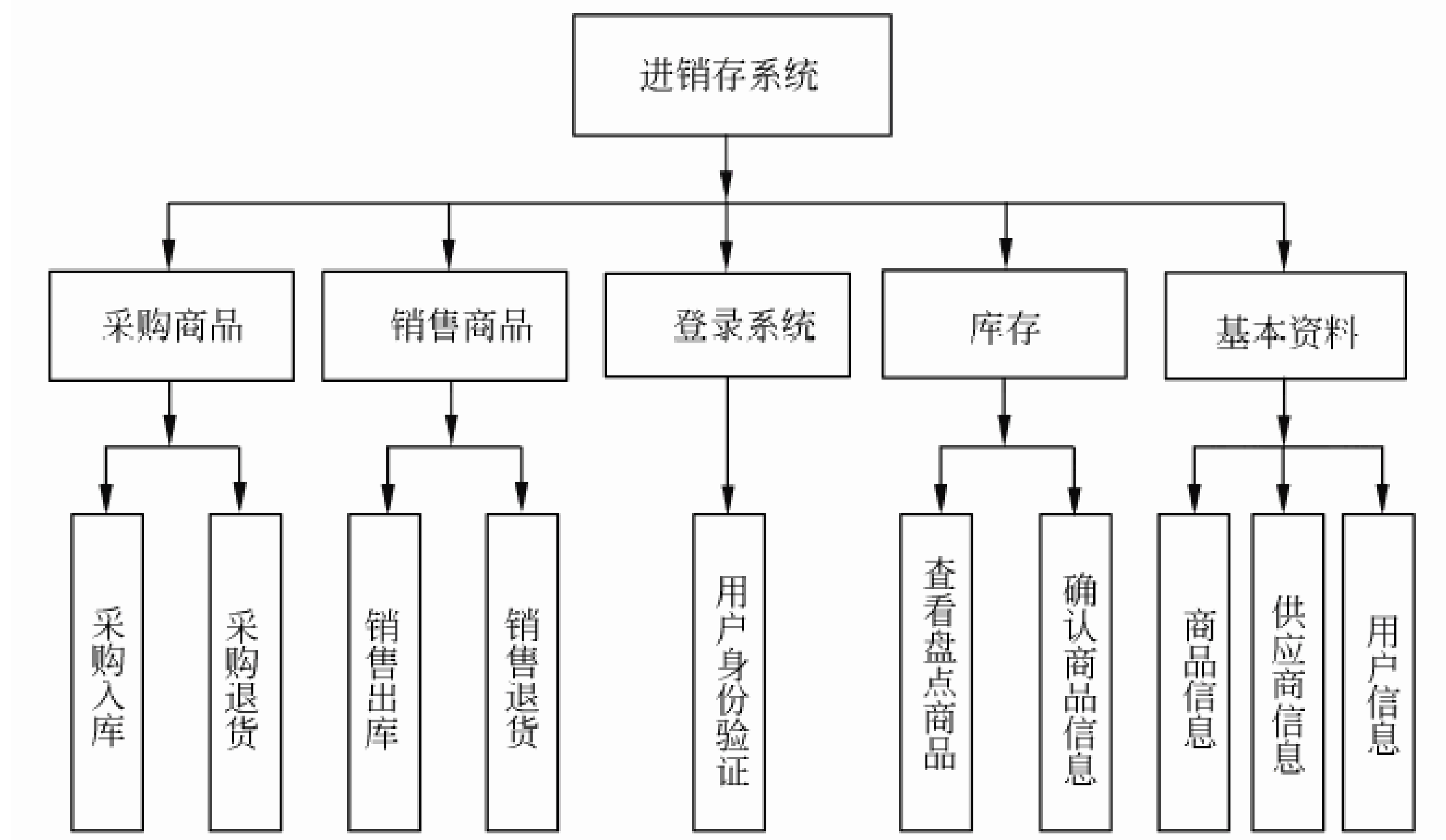


图 19-9　进销存系统模块结构图

19.3　系 统 设 计

系统分析完成后，进入系统设计阶段，这是整个系统实现过程中非常重要的一个阶段。软件设计的主要任务是把需求分析得到的结果转换为软件结构和数据结构，建立目标系统的逻辑模型，从而形成系统架构。明确软件系统应该“怎样做”。

1. 概要设计

（1）软件结构设计：将一个复杂系统按功能进行模块划分、建立模块的层次结构及调用关系、确定模块间的接口及人机界面等。

（2）数据结构设计：数据特征的描述，确定数据的结构特性及数据库的设计。

2. 详细设计

（1）为每个模块确定采用的算法，选择某种适当的工具表达算法的过程，写出模块的详细过程性描述。

（2）确定每一模块使用的数据结构。

（3）确定模块接口的细节，包括对系统外部的接口和用户界面，对系统内部其他模块的接口，以及模块输入数据、输出数据及局部数据的全部细节。

（4）要为每一个模块设计出一组测试用例，以便在编码阶段对模块代码（即程序）进行预定的测试。

19.3.1　数据库逻辑结构设计

数据库设计是系统设计中非常重要的一个环节。数据库是一切系统设计的基础，通俗地说，数据库设计就像高楼大厦的根基一样，如果设计不合理、不完善，将在系统开发过程中，甚至到后期系统的维护、功能的变更和功能扩充时引起较多的问题，严重时甚至要重新设计项目，重做大量已完成的工作。

根据功能模块划分的结果可知，本系统的用户主要是商品销售人员、商品采购人员和库存管理人员。在使用系统时都需要先登录后才能使用在 PSS 系统中自己拥有权限的那部分功能，因此在本系统中需要创建存储用户信息的数据实体。除此之外，系统还需要用于存储商品信息、商品销售信息、商品库存信息、退货信息、订单信息和采购单信息，所以需要创建它们各自相应的数据实体。

- ❑ 用户数据实体：需要记录用户的登录名、真实姓名和密码，还有一个标识列，用于区分用户类型。登录名、密码和用户类型是登录系统时验证所必须的。
- ❑ 商品信息数据实体：包括商品名称、零售价格、批发价、大规格、小规格、倍率、品牌、厂家信息、出厂日期等。
- ❑ 订单数据实体：包括订单的添加时间、订单号、操作员、商品名称、数量和订单类型等。
- ❑ 库存数据实体：包括商品名称、数量和库存提示。
- ❑ 退货数据实体：包括商品名称、数量、原因、操作员、时间和厂家编号。

任何一个系统中存储和管理的对象都不是相互独立的，都有一定的联系和关联，分析清楚关系有助于分析业务和数据流向。那么采购员和商品的关系就很明确了——进货和商

品（一对多），商品和供应商的关系——提供（一对多），商品和品牌的关系——属于（一对一）。

注意：数据设计的一般步骤是收集信息→标识对象→标识对象的属性→标识对象间的关系。

在设计数据库表的过程中，一般要遵循以下几条原则。

- 数据库的一个表最好只存储一个实体或对象的相关信息，不同的实体最好存储在不同的数据表中，如果实体还可以再划分，要掌握实体的划分原则是最好能够比当前系统要开发的实体的复杂度小。
- 数据表的信息结构一定要合适，表的字段数量一般不要太多。
- 扩充信息和动态变化的信息，一定要分别放在不同的表里。

19.3.2　系统 E-R 图

通过上面的数据库逻辑结构设计分析，最后要对分析的结果进行评估，那么稍微复杂一点的系统用文字描述就显得不够清晰了，这时就需要使用通用的实体关系图，也就是 E-R 图来表达了。结合上述分析，构建出的 E-R 图如图 19-10 所示。

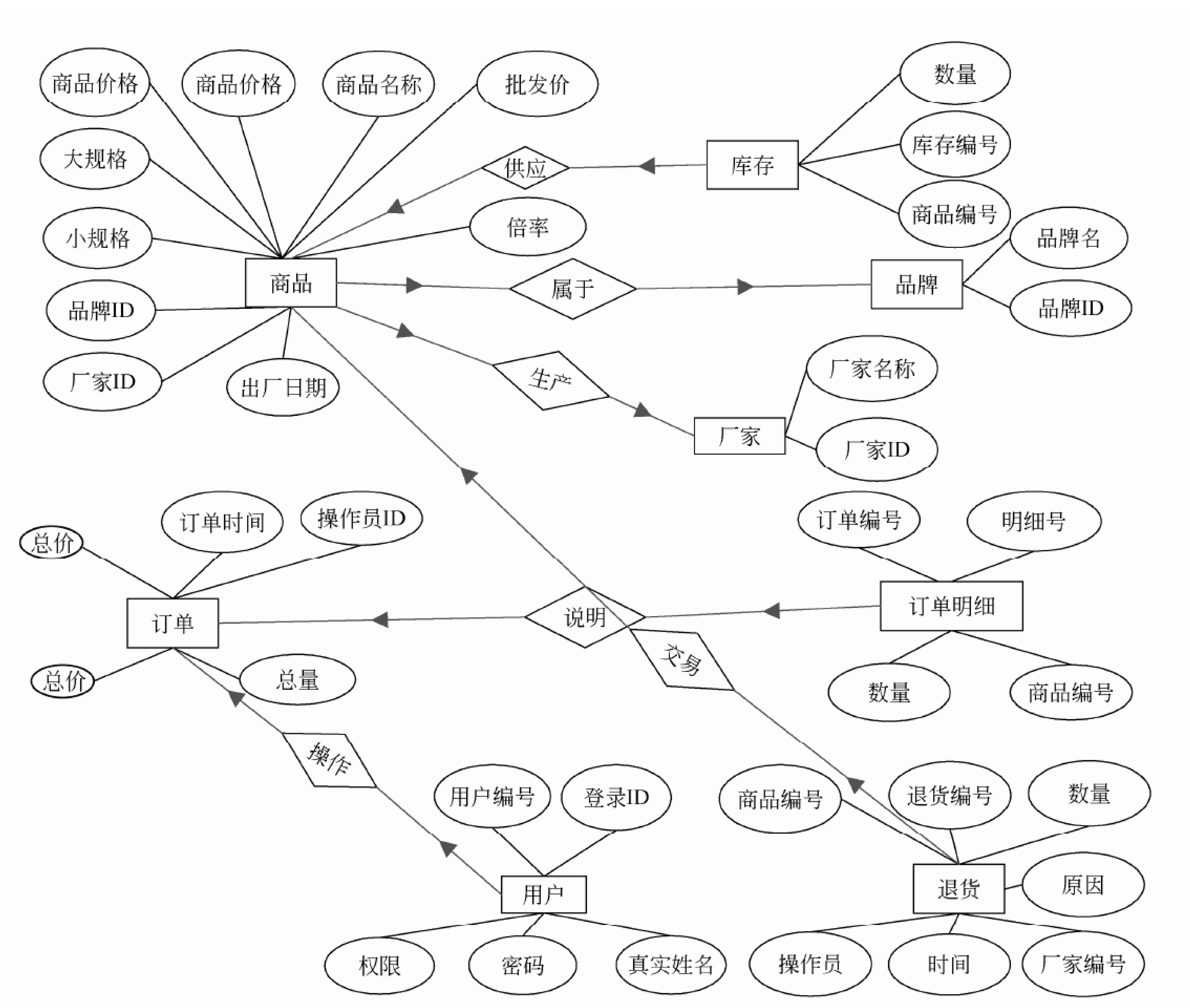


图 19-10　系统实体关系图

🔔注意：绘制 E-R 图的软件有很多种，常用的软件有 PowerDesigner、Visio。

19.3.3　数据库逻辑设计

数据库的逻辑设计已经完成，接下来就是逻辑实现阶段，这个阶段需要注意的是如何使用三大范式约束逻辑设计。一般遵循以下 3 个原则即可。

- 每个字段都必须是不可再分的。
- 非主键字段必须完全依赖该主键。
- 有关联关系的表使用主外键约束起来。

最终形成翻译后的数据库关系图，如图 19-11 所示。

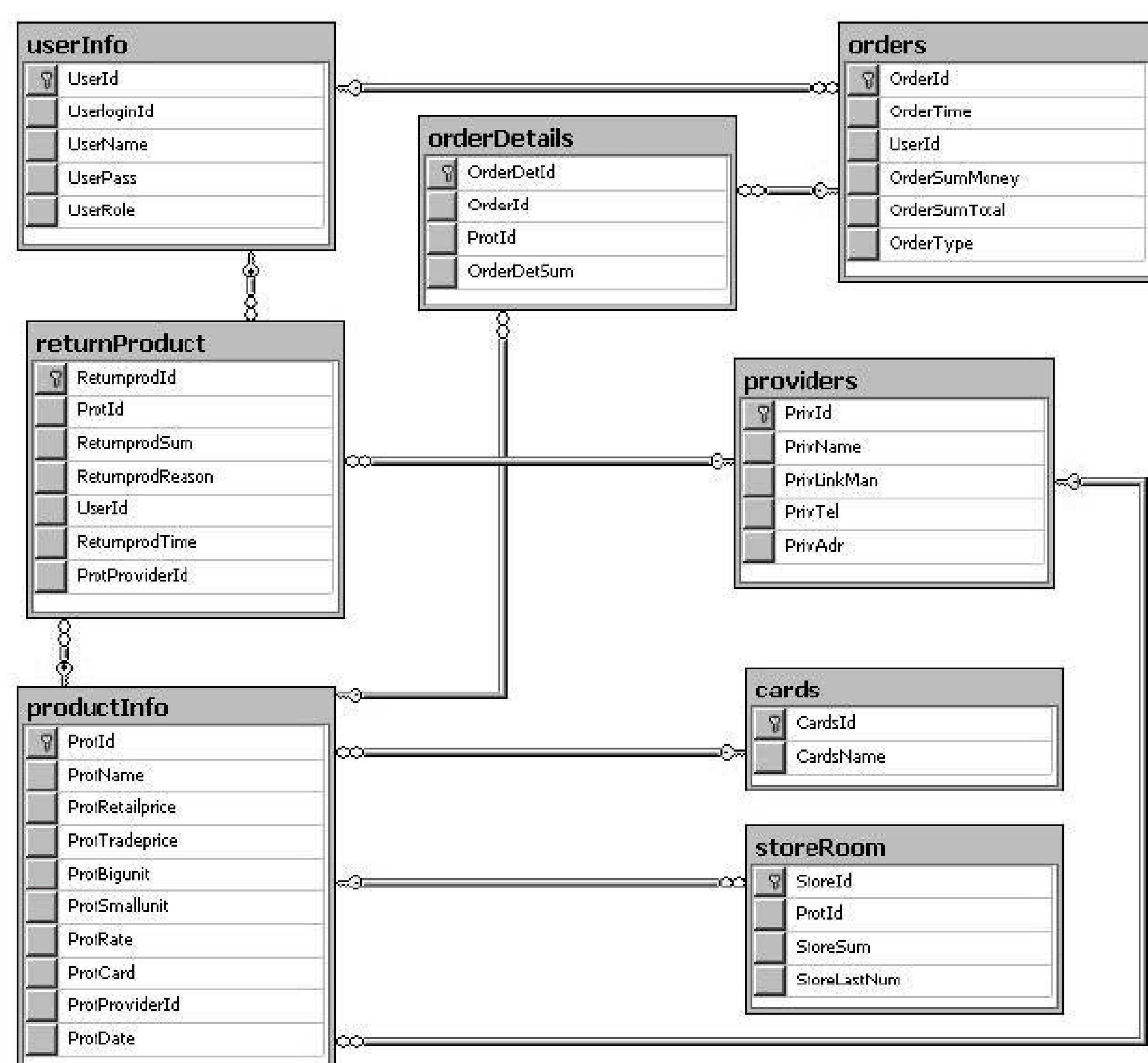


图 19-11　数据库关系图

19.3.4　表设计

由于本案例采用的是 SQL Server 2008 数据库，就需要按照 SQL Server 2008 的语法和规范把 19.3.3 的数据库逻辑设计反映成为物理设计，即创建物理表的过程，需要确定每列的数据类型、长度和约束。另外，每个字段是否可以为空、长度有没有限制等，均需要和具体需求对照确定。约定数据库名为 PSSDB，具体表设计如表 19-3 所示。

表 19-3　进销存信息管理系统数据库PSSDB的表定义

序　　号	字　　段	类　　型	备　　注
UserInfo（用户表）	UserId	Int	用户编号，主键，自增
	UserLoginId	Varchar(20)	登录 ID，不为空
	UserName	Varchar(20)	真实姓名，不为空
	UserPass	Varchar(20)	密码，不为空
	UserRole	Int	权限，不为空

续表

序　　号	字　　段	类　　型	备　　注
ProductInfo（商品表）	ProtId	Int	商品编号，主键自增
	ProtName	Varchar(20)	商品名称，不为空
	ProtRetailPrice	Money	零售价格，不为空
	ProtTradePrice	Money	批发价，不为空
	ProtBigunit	Varchar(10)	大规格，不为空
	ProtSmallunit	Varchar(10)	小规格，不为空
	ProtRate	Int	倍率，不为空
	ProtCard	Int	外键，品牌 ID，不为空
	ProtProviderId	Int	外键，厂家 ID，不为空
	ProtDate	Date	出厂日期，不为空
Cards（品牌表）	CardsId	Int	品牌 ID，主键，自增
	CardsName	Varchar(20)	品牌名，不为空
Providers（供货商表）	PrivId	Int	厂家 ID，主键，自增
	PrivName	Varchar(20)	厂家名称，不为空
	PrivLinkMan	Varchar(20)	联系人，不为空
	PrivTel	Varchar(20)	电话，不为空
	PrivAdr	Varchar(50)	地址，不为空
Orders（订单表）	OrderId	Int	订单编号，主键，自增
	OrderTime	Datetime	订单时间，不为空
	UserId	Int	外键，操作员 ID，不为空
	OrderSumMoney	Money	总价，不为空
	OrderSumTotal	Int	总量，不为空
	OrderType	Int	类型：0 进货　1 退货　2　销售　3 销售退货，不为空
OrderDetails（订单明细表）	OrderDetId	Int	明细编号，主键，自增
	OrderId	Int	订单号，外键，不为空
	ProtId	Int	商品编号，外键，不为空
	OrderDetSum	Int	数量，不为空
StoreRoom（库存表）	StoreId	Int	库存编号，主键，自增
	ProtId	Int	商品编号，外键，不为空
	StoreSum	Int	数量（小包装），不为空
	StoreLastNum	Int	库存警告，不为空
ReturnProduct（退货表）	ReturnprodId	Int	退货编号，主键，自增
	ProtId	Varchar（20）	商品编号，外键，不为空
	ReturnprodSum	Int	数量，不为空
	ReturnprodReason	Varchar（50）	原因，不为空
	UserId	Int	操作员，外键，不为空
	ReturnprodTime	Datetime	时间，不为空
	ProtProviderId	Int	厂家 ID，外键，不为空

注意：这里的数据类型和长度是根据实际调查和与客户沟通确定的，包括专有名词也从现实中来，再一次说明了需求的重要性。

19.4　进销存管理系统界面设计

一个好的应用程序不但要有完善的功能，还要有优雅的用户使用界面。用户界面是应用程序的一个重要组成部分。用户界面不仅影响到软件外观，而且对应用程序的易用性和可操作性都起了决定性作用。界面的设计应该从用户角度出发，以方便用户使用作为根本目标，因为整个软件使用过程中，界面和用户交互的时间是最长的。

19.4.1　界面设计标准

由于软件界面在整个软件生命周期内的重要作用，所以必须根据社会工程学、国家标准等相关规范，确定软件界面时要遵循以下原则：

- 布局合理，界面清洁。
- 配色合理，图像和显示效果要统一。
- 整个软件界面风格应该保持一致。
- 减少用户的操作负担，界面尽可能少地使用鼠标。
- 所有界面文字清晰明了，不产生歧义。

19.4.2　系统界面操作流程

按照一般的操作习惯，本项目涉及的界面及界面间的关系，可以用图 19-12 表示。主要包括：界面外观和布局、用户操作流程、界面输入和输出、界面对应的逻辑操作等。

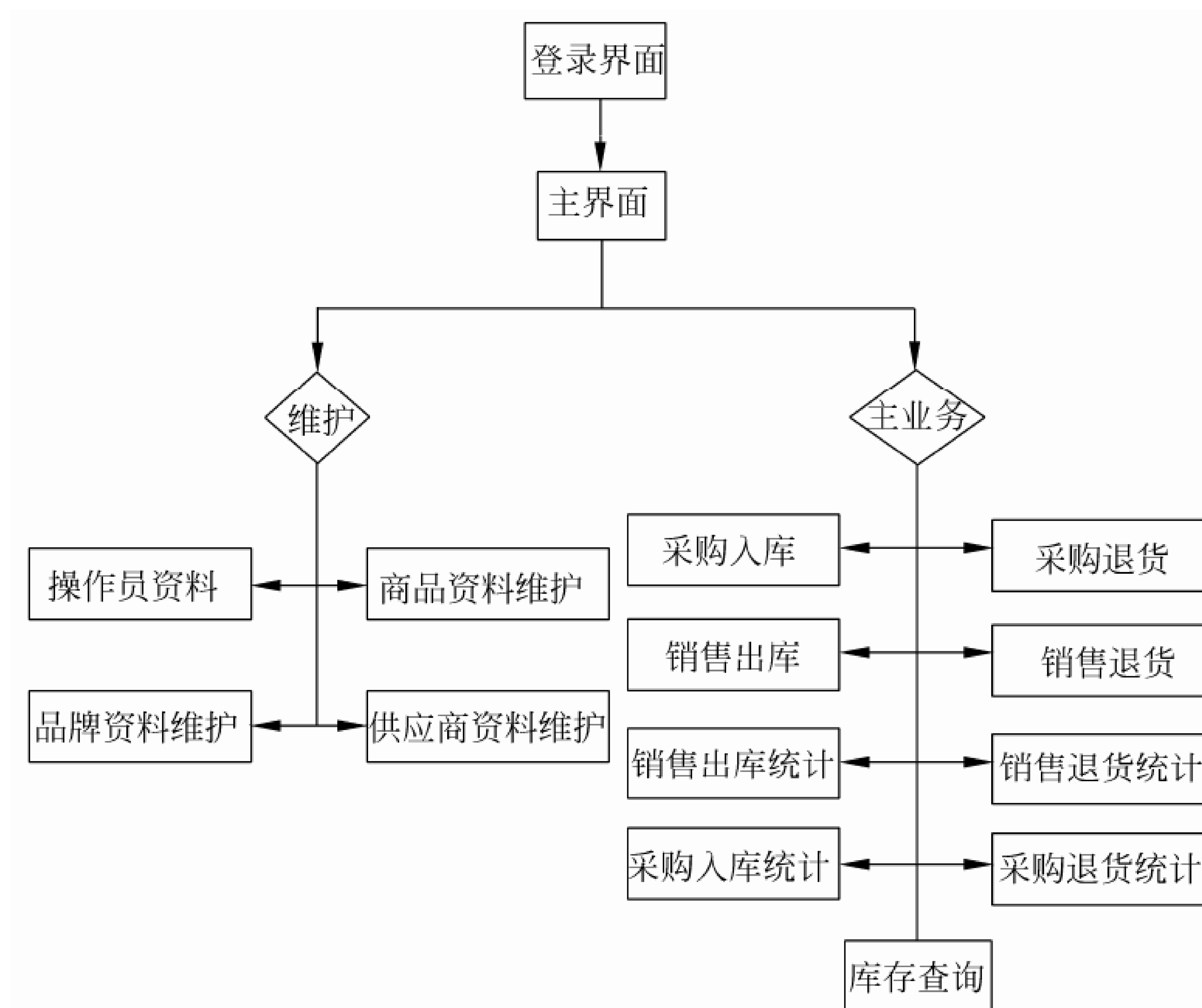


图 19-12　用户操作界面流程图

由于进销存信息管理软件业务本身不由界面决定，所以这里就不为每个界面进行外观和布局上的设计，但是还需要更详细地定义每个界面的具体功能及它所需要显示或操作的数据。页面信息管理中，各个界面功能的具体定义如表 19-4 所示。

表 19-4　进销存系统软件界面功能定义

界　面	功　能	相关数据
登录界面	提供登录系统的功能，操作员输入用户名、密码，选择登录类型，单击登录。登录失败提示，否则跳转到主操作界面	登录名输入框、密码输入框、登录类型下拉列表框、登录按钮、退出按钮
主界面	为操作员提供各功能的进入菜单	菜单栏包括的菜单项：基本资料维护、采购、销售、库存和统计查询。显示当前登录用户信息及系统当前时间的状态栏
操作员资料维护界面	提供对操作员添加、删除、修改、查询功能。该界面只有管理员有权进入	工具条：图片按钮、文本框 用于显示用户信息的控件 DataGridView 用户增加、修改用户信息的：文本框
商品资料维护界面	提供对商品的添加、删除、修改、查询功能。该界面只有管理员和采购员有权进入。（添加操作提示：新添加的商品库存数量为 0）	工具条：图片按钮、文本框 用于显示商品信息的控件：DataGridView 用于商品增加，修改商品信息的：文本框
供应商资料维护界面	提供对供应商的添加、删除、修改、查询功能。该界面只有管理员和采购员有权进入	工具条：图片按钮、文本框 用于显示供应商信息的控件：DataGridView 用于供应商增加，修改供应商信息的：文本框
品牌资料维护界面	提供对商品品牌的添加、删除、修改、查询功能。该界面只有管理员和采购员有权进入	工具条：图片按钮、文本框 用于显示品牌信息的控件：DataGridView 用于品牌增加，修改品牌信息的：文本框
采购入库界面	提供采购商品入库。当光标进入商品名称文本框时，按下 Enter 键，选择要采购的商品。按下 Enter 键，输入采购数量，按下 Enter 键，之后单击保存图片按钮。提示采购入库成功，否则提示入库失败！（采购提示：当采购的商品是库存中所没有的新商品时，要先到商品维护界面添加商品信息，之后才能对新商品进行采购。） 该界面只有管理员和采购员有权进入	工具条：图片按钮、文本框 用于显示所采购的商品的信息的控件：DataGridView 用于商品数量增加和采购信息显示的：文本框
采购退货界面	提供采购商品退货。当光标进入商品名称文本框时，按下 Enter 键，选择要退货的商品。按下 Enter 键，输入退货数量，按下 Enter 键，之后单击保存图片按钮，弹出退货原因窗体输入退货原因后。提示退货成功，否则提示退货失败！（退货提示：当退货的商品数量大于该退货商品的库存数量时，提示库存不够。） 该界面只有管理员和采购员有权进入	工具条：图片按钮、文本框 用于显示退货商品的信息的控件：DataGridView 用于商品数量和退货信息显示的：文本框

续表

界　面	功　能	相 关 数 据
销售出库界面	提供销售商品。当光标进入商品名称文本框时，按下 Enter 键，选择要销售的商品。按下 Enter 键，输入销售数量，按下 Enter 键，之后单击保存图片按钮。提示销售成功，否则提示销售失败！（销售提示：当销售的商品数量小于该商品的库存警报数量时，提示，该商品已经到达销售警报位置，需要进货） 该界面只有管理员和销售员有权进入	工具条：图片按钮、文本框 用于显示销售商品的信息的控件：DataGridView 用于商品数量和销售信息显示的：文本框
销售退货界面	提供销售商品退货。当光标进入商品名称文本框时，按下 Enter 键，选择要退货的商品。按下 Enter 键，输入退货数量，按下 Enter 键，之后单击保存图片按钮。提示退货成功，否则提示退货失败！ 该界面只有管理员和销售员有权进入	工具条：图片按钮、文本框 用于显示退货商品的信息的控件：DataGridView 用于商品数量和退货信息显示的：文本框
库存查询界面	提供商品信息库存查询功能	工具条：图片按钮、文本据 显示库存商品的信息的控件：DataGridView
销售出库统计	提供商品销售出库的记录查询。该界面只有管理员和销售员有权进入	用于显示商品销售出库记录的详细信息控件：DataGridView 显示订单详细信息的控件：按钮
销售退货统计	提供商品销售退货的记录查询。该界面只有管理员和销售员有权进入	用于显示商品销售退货记录的详细信息控件：DataGridView 显示订单详细信息的控件：按钮
采购入库统计	提供商品采购入库的记录查询。该界面只有管理员和采购员有权进入	用于显示商品采购入库记录的详细信息控件：DataGridView 显示订单详细信息的控件：按钮
采购退货统计	提供商品采购退货的记录查询。该界面只有管理员和采购员有权进入	用于显示商品采购退货记录的详细信息控件：DataGridView 显示订单详细信息的控件：按钮

19.5　进销存管理软件的具体实现

前面的设计可以帮助我们了解整个系统的结构，本节将进入正式的开发阶段，如果读者还不明白这个系统能做什么，一定要仔细阅读前面的知识。

19.5.1　解决方案资源管理器

在程序设计和开发的前期工作都完成的情况下，接下来就是具体的实现阶段，也就是

软件编码。软件编码就是将上一阶段的详细设计到的对处理过程的描述，转换为基于某种计算机语言（这里使用的计算机语言是 C#）的程序，即源程序代码。

按照进销存管理信息系统的需求，本项目需要有客户操作界面，由于在局域网内使用，所以首选.NET 平台下大家都熟悉的 WinForm 界面应用程序，专门负责和操作员交互，接收输入和显示输出。

搭建项目：打开 Visual Studio 2010 新建项目，在新建项目的模板中选择 Windows 窗体应用程序。在用于创建具有 Windows 窗体用户界面的应用程序的项目（.NET Framework 4.0）下填写项目名称和项目储存位置，单击“确定”按钮后项目就创建了。这里约定项目名称为 DLAPSS。

在进行编码前，首先将 19.4 节分析得到的进销存系统里应该具备的操作界面，在解决方案中先设计好，通俗地说就是画界面。在实际开发 Windows 窗体应用程序项目时，一般都会先把界面设计好，然后再写具体实现的代码。在进销存系统的解决方案中，设计好的 Windows 窗体用户界面目录如图 19-13 所示。

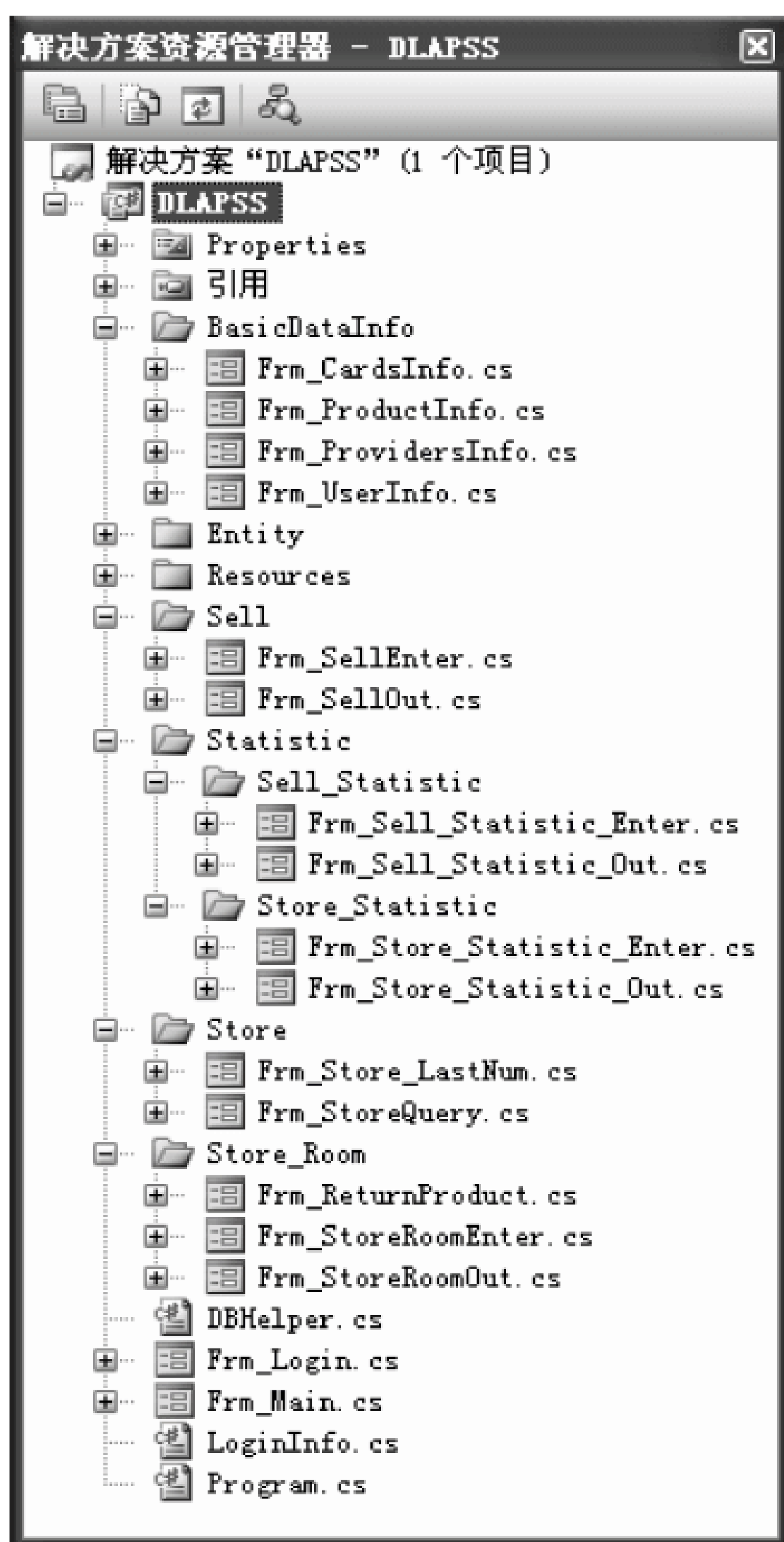


图 19-13　进销存系统目录结构图

在这个目录结构中，DLAPSS 是项目名称。其下文件夹 Resources 是用于存放项目中用到的图片；文件夹 Entity 是用于存放实体类；文件夹 BasicDataInfo 存放的是基本资料维护操作界面；文件夹 Sell 存放有关销售的界面；文件夹 Sell_Statistic 存放的是有关销售统计的界面；文件夹 Store_Statistic 存放的是有关采购统计界面；文件夹 Store 存放库存查询界面。

文件夹 Store_Room 存放的是有关采购的界面。这个目录结构读者可以根据自己的需要进行设计。

19.5.2　定义 DBHelper

本系统采用 ADO.NET 技术进行开发，为了更好地进行对数据库的相关操作，这里定义一个 DBHelper 类存放数据库的连接字符串。

在实现某一功能时，首先要连接数据库。那么，如果每次需要用到数据库时，都需要重写一遍数据库连接字符串，这样不但很麻烦而且当系统一旦移植数据库访问参数改变，就需要对写过的每一段连接字符串都进行改写。

鉴于此，使用一个 DBHelper 类封装数据库连接字符串。具体代码如示例代码 19-1 所示。

示例代码 19-1

```
public static class DBHelper
{
    //连接数据库字符串
```

```
    public static readonly string conStr = "Server=.;Database=PSSDB;Uid=
    sa;Pwd=5201314";
 }
```

这样一来，在数据库移植或系统环境改变时，只需修改这个地方就可以了，十分方便。读者可以在这个基础上进行进一步改进，比如将对数据库进行增删改操作的方法抽出来等。

19.5.3　用户身份验证功能

用户登录系统时，系统会对其身份进行验证。如果系统中不存在该用户，会提示用户名或密码错误！否则系统会根据不同登录类型让其拥有不同的对系统访问权限。在进行用户登录信息验证时，需要操作数据库，因此需引入两个命名空间。登录时对用户身份进行验证的具体代码如示例代码 19-2 所示。

示例代码 19-2

```
//引用的命名空间
using System.Data;
using System.Data.SqlClient;
//验证登录方法
public void loginValidate()
{
    if (txt UserloginId.Text.Trim() == "" || string.IsNullOrEmpty(txt
    UserloginId.Text))
    {
        MessageBox.Show("用户名不能为空！", "登录提示");
        //将用户名文本框设置为输入焦点
        txt UserloginId.Focus();
    }
    else if (txt UserPass.Text.Trim() == "" || string.IsNullOrEmpty(txt
    UserPass.Text))
    {
        MessageBox.Show("密码不能为空！", "登录提示");
        //将密码文本框设置为输入焦点
        txt UserPass.Focus();
    }
    else if (cbo loginType.Text.Trim() == "" || string.IsNullOrEmpty(cbo
    loginType.Text.Trim()))
    {
        MessageBox.Show("请选择登录类型！", "登录提示");
    }
    else
    {
        UserInfo u = null;
        SqlConnection con = new SqlConnection(DBHelper.conStr);
                                                //获取数据库连接对象
        try
        {
            con.Open();                         //打开数据库连接
            int userRole = cbo loginType.SelectedIndex;
            string sql = string.Format("select * from userInfo where" +
            " UserloginId='{0}'and UserPass='{1}' and UserRole={2}",
            txt UserloginId.Text, txt UserPass.Text, userRole);
            SqlCommand com = new SqlCommand(sql, con);
                                                //获取命令执行对象
```

```
            SqlDataReader dr = com.ExecuteReader();
                                                     //获得数据读取对象
            if (dr.Read())
            {
                u = new UserInfo();                  //将用户登录信息保存到对象中
                u.UserId = Convert.ToInt32(dr["UserId"]);
                //由于篇幅的限制，详细代码可参考光盘源代码
                …
            }
            dr.Close();                              //关闭数据库读写对象
        }
        catch (Exception)
        {
            MessageBox.Show("系统出现异常，请您稍后再试！", "登录提示");
        }
        finally
        {
            con.Close();                             //关闭数据库连接
        }
        if (u != null)
        {
            LoginInfo.LoginUserInfo = u;             //保存登录用户信息
            this.Visible = false;
            Frm_Main fm = new Frm_Main();
            fm.Show();                               //打开主操作界面
        }
        else
        {
            MessageBox.Show("用户名或密码错误！", "登录提示");
        }
    }
}
```

提示：上面代码中的 cbo_loginType.Text.Trim()中的 Trim()方法用于去除字符串两端的空格，代码 string.IsNullOrEmpty(cbo_loginType.Text.Trim())用于验证非空，值得注意的是当字符串为 null 时返回的是 true。

为方便用户操作，在用户选择了登录类型后直接按下 Enter 键，就可以跳转到主操作界面。实现按下 Enter 键跳转到主操作界面的具体代码如示例代码 19-3 所示。

示例代码 19-3

```
private void cbo_loginType_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter 键)                    //当按下 Enter 键时
    {
        if (txt_UserloginId.Text != "" && txt_UserPass.Text != "")
          loginValidate();                           //调用验证登录方法
        else
            SendKeys.Send("{TAB}");
    }
}
```

拓展提示：上面这段代码实现的是如果用户按下的是 Enter 键，判断用户的输入是否完整，如果完整则调用登录验证方法，进行登录验证操作，否则让按下 Enter 键的效果相当于按下 Tab 键。

19.5.4　主界面的功能

当用户进入主操作界面后，可以选择属于他这个角色所能操作的菜单。单击菜单项后会跳转到相应的窗体。在进销存系统中主窗体为 MDI 窗体，在其下所打开的窗体都会在关闭主窗体之后关闭。要实现让主窗体为 MDI 窗体，首先需要设置主窗体的 IsMdiContainer 属性为 True，再指定子窗体的父窗体是主操作界面窗体（MDI 窗体）。其具体实现代码如示例代码 19-4 所示。

示例代码 19-4

```
private void tsmi_StoreRoomOut_Click(object sender, EventArgs e)
{
    Frm_StoreRoomOut fsro = new Frm_StoreRoomOut();
    fsro.MdiParent = this;      //指定父窗体
    if (fsro.Visible)
    {
       fsro.Focus();             //设置为焦点
       return;                   //程序跳出
    }
    else
    {
       fsro = new Frm_StoreRoomOut();
       fsro.MdiParent = this;
       fsro.Show();
    }
}
```

在主窗体的状态条上显示当前登录的用户名和系统当前时间，窗体加载时发生，也就是在窗体的 Load 事件下发生。具体实现代码如示例代码 19-5 所示。

示例代码 19-5

```
//timer 下的事件，每隔 1 秒刷新一次系统时间
private void timer_Tick(object sender, EventArgs e)
{
    //将日期格式化为：yyyy 年 MM 月 dd 日 HH 时:mm 分:ss 秒格式
    tslb_time.Text = DateTime.Now.ToString("yyyy 年 MM 月 dd 日 HH 时:mm 分:ss
    秒");
}
//窗体加载时发生
private void Frm_Main_Load(object sender, EventArgs e)
{
    if (LoginInfo.LoginUserInfo.UserRole == "1")   //当前登录用户是销售员
    {
       tsmi_BasicDataInfo.Enabled = false;
       tsmi_Store_Room.Enabled = false;
       tsmi_Store_Statistic.Enabled = false;
    }
    else if (LoginInfo.LoginUserInfo.UserRole == "2")
                                                   //当前登录用户是采购员
    {
       tsmi_UserInfo.Enabled = false;
       tsmi_Sell.Enabled = false;
```

```
        tsmi Sell Statistic.Enabled = false;
    }
    //在状态栏显示欢迎信息
    tslb_name.Text = "登录信息：欢迎您！" + LoginInfo.LoginUserInfo.UserName
    + "！当前时间：";
    tslb_time.Text = DateTime.Now.ToString("yyyy年MM月dd日HH时:mm分:ss
    秒");
}
```

注意：代码 DateTime.Now.ToString("yyyy 年 MM 月 dd 日 HH 时:mm 分:ss 秒")实现了获取系统当前时间的功能，以指定的字符串格式显示。窗体加载事件下，根据登录时，存储的用户信息判断当前登录用户的类型，分配相应的权限。

19.5.5　商品资料维护功能

1．查询商品所有信息

当单击“商品资料维护”选项后，在商品资料维护窗体加载时应该将所有的商品信息都显示出来。使用列表控件展示概要信息，附加文本框和下拉列表框显示详细信息，当用户单击列表时需要显示详细信息，具体实现代码如示例代码 19-6 所示。

示例代码 19-6

```
//窗体加载
private void Frm_ProductInfo_Load(object sender, EventArgs e)
{
    List<ProductInfo> lp = new List<ProductInfo>();
    dgv PrdouctInfo.AutoGenerateColumns = false;   //设置只显示手动编辑的列
    lp =GetProductInfoByProt_Name("");      //调用查询商品所有信息的方法
    dgv_PrdouctInfo.DataSource = lp;        //指定 dgv_PrdouctInfo 的数据源
    Conceal();                              //调用隐藏控件的方法
    cbb_Cards_Name.DisplayMember = "Cardsname";
                                       //设置下拉列表框的显式值为 Cardsname
    cbb Cards Name.ValueMember = "Cardsid";
                                            //设置下拉列表框的隐式值为 Cardsid
    cbb Cards Name.DataSource =LoginInfo.GetCardsByCards name("");
    cbb Priv Name.DisplayMember = "Priv name";
    cbb_Priv_Name.ValueMember = "Priv_id";
    cbb_Priv_Name.DataSource = LoginInfo.GetProvidersByPriv_name("");
                                            //设置数据源
}
//查询商品所有信息
public List<ProductInfo> GetProductInfoByProt Name(string Prot name)
{
    List<ProductInfo> lp = new List<ProductInfo>();
                                            //声明一个用于存放商品信息集合
    SqlConnection con = new SqlConnection(conStr);
                                            //获取数据库连接对象
    try
    {
        con.Open();                         //打开数据库链接
        string sql = string.Format("select productInfo.*, Cards.*,provid-
        ers.* from productInfo,Cards,providers where productInfo.Protcard=
```

```
        Cards.Cardsid and productInfo.ProtproviderId=providers.Privid and
        ProtName like '%{0}%'", Prot name);
        SqlCommand com = new SqlCommand(sql, con);
                                            //创建 SqlCommand 数据操作对象
        SqlDataReader dr = com.ExecuteReader();
                                            //创建 SqlDataReader 数据读取对象
        while (dr.Read())
        {
            ProductInfo p = new ProductInfo();  //创建商品信息对象
            p.Prot id = Convert.ToInt32(dr["Protid"]);
                                            //将读出的 ID 强制转换为整形
            //由于篇幅的限制，详细代码请参考光盘源代码
            …
            CardsInfo loadCards = new CardsInfo();
            loadCards.CardsId = Convert.ToInt32(dr["CardsId"]);
            loadCards.CardsName = dr["CardsName"].ToString();
            lp.Add(p);                      //将对象添加到集合中
        }
        dr.Close();                         //关闭数据读取对象
    }
    catch(Exception)
    {
        MessageBox.Show("系统出现异常，请您稍后再试", "登录提示");
    }
    finally
    {
        con.Close();                        //关闭链接
    }
    return lp;
}
```

2．添加/修改商品信息

单击“保存”图片按钮时，系统会根据当前的操作标识判断执行的是添加还是修改，之后再执行相应的操作。按照一般的管理类软件做法，不管新增还是修改按钮，单击之后都不发生实际的数据库操作，而是再次单击“保存”按钮之后才会发生。其具体实现代码如示例代码 19-7 所示。

示例代码 19-7

```
// 修改添加保存
private void Save()
{
    int Prot id = Convert.ToInt32(lb Prot id.Text);
    string Prot name = txt Prot Name.Text;
    float Prot retailprice = ConvertToSingle(txt Prot Retailprice.Text);
    float Prot_tradeprice = Convert.ToSingle(txt_Prot_Tradeprice.Text);
    string Prot Biunit = txt Prot Bigunit.Text;
    string Prot smallunit = txt Prot Smallunit.Text;
    int prot rate = Convert.ToInt32(txt Prot Rate.Text);
    int Prot_cardId = Convert.ToInt32(cbb_Cards_Name.SelectedValue);
                                                //获取下拉值
    int Prot providerId = Convert.ToInt32(cbb Priv Name. SelectedValue.
    ToString());
    DateTime Prot_date = dtp_Prot_Date.Value;
```

```
    string Store LastNum = txt Store LastNum.Text;
    if (type == 0)
    {
        foreach (ProductInfo item in lp)          //遍历集合查询是否存在
        {
            if (item.Prot_name.Equals(txt_Prot_Name.Text))
            {
                MessageBox.Show("该商品已存在！","提示");
                return;
            }
        }
        string sql = string.Format("insert into productInfo(ProtName,"
                 +" ProtRetailprice,ProtTradeprice,ProtBigunit, ProtSmal-
                 lunit,"+" ProtRate,ProtCard,ProtProviderId,ProtDate)"
                 + "values('{0}','{1}','{2}','{3}','{4}',{5},{6},'{7}',
                 '{8}')", Prot name, Prot retailprice, Prot tradeprice,
                 Prot_Bigunit, Prot_smallunit, prot_rate, Prot_cardId,
                 Prot providerId, Prot date);
                 Int result = ExecuteQuery(sql);    //执行插入操作
                 if (result > 0)
                 {
                     SqlConnection con = new SqlConnection(conStr);
                     con.Open();                    //打开数据库连接
                     SqlCommand cmd = new SqlCommand( "select max(protid)
                     from productInfo",con);
                     int inputProtId = Convert.ToInt32(cmd. ExecuteSca-
                     lar());
                     con.Close();                   //关闭连接
                     if (inputProtId != 0)
                     {
                         string sql2 =
                         string.Format("insert into storeRoom(Protid,
                         storesum, storelastNum)values({1},0,{0})", Convert.
                         ToInt32(txt Store LastNum.Text), inputProtId);
                         result = result + ExecuteQuery(sql2);
                         MessageBox.Show("添加成功！", "提示");
                      }
                  }
    }
    else if (type == 1)
    {
        string sql = string.Format("update productInfo set ProtName='{0}',"
                  +"ProtRetailprice='{1}',ProtTradeprice='{2}', ProtBigu-
                  nit='{3}'," +"ProtSmallunit='{4}',ProtRate={5},ProtCa-
                  rd={6},ProtProviderId={7},"+"ProtDate='{8}'where ProtId
                  ={9}", Prot_name, Prot_retailprice, Prot_tradeprice,
                  Prot Bigunit, Prot smallunit, prot rate, Prot cardId,
                  Prot providerId, Prot date, Prot id);
         int result = ExecuteQuery(sql);            //执行更新操作
         if (result > 0)
         {
             MessageBox.Show("修改成功！", "提示");
         }
    }
}
```

3．删除商品信息

当在商品列表中选中商品后，单击删除图片按钮。系统会提示你确定要删除此商品信息吗？删除之后所有关于该商品的记录将会全部清空！单击确定后，系统提示删除成功或该商品还有库存，请先将货物清理出库，再删除商品！其具体实现代码如示例代码 19-8 所示。

示例代码 19-8

```
//删除事件
private void tsb_Btn_Delete_Click(object sender, EventArgs e)
{
    if (dgv_PrdouctInfo.SelectedRows.Count > 0)
    {
        int pId = Convert.ToInt32(dgv_PrdouctInfo.SelectedRows[0]. Cells
         [0]. Value);
        string sql = "delete from productInfo where ProtId=" + pId;
        DialogResult dr = MessageBox.Show("你确定要删除此商品信息吗？删除之后所
        有关于该商品的记录将会全部清空", "提示",MessageBoxButtons.OKCancel);
        string sqlStoreCount="select StoreSum from storeRoom where
        protid="+pId;
        int storeCount=0;
        SqlConnection con = new SqlConnection(conStr);
        con.Open();
        SqlCommand cmd = new SqlCommand(sqlStoreCount, con);
        storeCount=Convert.ToInt32(cmd.ExecuteScalar());
        if (storeCount > 0)
        {
            MessageBox.Show("该商品还有库存，请先将货物清理出库，再删除商品！",
            "提示");
            return;
        }
        else
        {
            string sqlDelStoreCount = "delete from storeRoom where protid
            ="+pId;
            string sqlDelDetial = " delete from orderdetails where protid="
            + pId;
            string sqlDelReturn = "delete from returnproduct where protid
            ="+pId;
            //调用执行增删改操作的方法
            ExecuteQuery(sqlDelStoreCount);
            ExecuteQuery(sqlDelDetial);
            ExecuteQuery(sqlDelReturn);
        }
        con.Close();
        if (dr == DialogResult.OK)
        {
            int result = ExecuteQuery(sql);
            if (result > 0)
            {
                MessageBox.Show("删除成功!", "提示");
                lp = GetProductInfoByProt Name("");
                dgv PrdouctInfo.DataSource = lp;
            }
        }
    }
```

```
}
//执行增删改操作的方法
public int ExecuteQuery(string sql)
{
    int i = 0;
    SqlConnection con = new SqlConnection(conStr);//创建数据连接对象
    try
    {
        con.Open();                          //打开数据库连接
        SqlCommand cmd = new SqlCommand(sql, con);
                                             //创建 SqlCommand 数据操作对象
        i = cmd.ExecuteNonQuery();           //执行增删改操作
    }
    catch (Exception)
    {
        MessageBox.Show("系统繁忙!", "提示");
    }
    finally
    {
        con.Close();                         //关闭数据库连接
    }
    return i;
}
```

4．商品信息查询

当用户在查询文本框中输入要查的商品名称后，单击查询图片按钮，就能看到要找的商品详细信息，否则提示要找的商品不存在。其具体实现代码如代码 19-9 所示。

示例代码 19-9

```
//查询商品详细信息
private void tsb Btn Lookup Click(object sender, EventArgs e)
{
    if (tsb_Txt_Lookup.Text.Trim() == "" || String.IsNullOrEmpty(tsb_Txt_
    Lookup.Text))
    {
        MessageBox.Show("请输入查询条件! ", "提示");
    }
    else
    {
        lp=GetProductInfoByProt_Name(tsb_Txt_Lookup.Text);
                                   //调用查询商品所有信息的方法
        dgv PrdouctInfo.DataSource = lp;
    }
}
```

5．实现按下Enter键相当于按下Tab键

为了方便用户的操作，进销存系统中所有的操作都可以按下 Enter 键完成，以提高软件的工作效率，此类做法常见于银行、邮局和火车站等系统中，这些系统对操作员的操作速度有着非常高的要求，同样作为商场进销存也是需要操作效率的。实现按下 Enter 键相当于按下 Tab 键的具体代码如示例代码 19-10 所示。

示例代码 19-10

```
private void txt Enter 键 KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter 键)
       SendKeys.Send("{TAB}");
}
private void txt Store LastNum KeyDown(object sender, KeyEventArgs e)
{
     if (e.KeyCode == Keys.Enter 键)
        Save();
}
```

19.5.6　供应商资料维护功能

1．查询供应商的资料

在供应商资料维护窗体加载时，保存按钮和文本框处于禁用状态，与此同时系统会将所有的供应商信息查询出来并显示。为方便用户查找特定供应商的详细信息，程序提供了根据供应商名称查询供应商详细信息的方法。其具体的实现代码如示例代码 19-11 所示。

示例代码 19-11

```
//窗体加载事件
private void Frm ProvidersInfo Load(object sender, EventArgs e)
{
    List<Providers> lp = new List<Providers>();    //存储厂商信息的集合
    tsb_Btn_Save.Enabled = false;
    Conceal();                                     //调用隐藏控件的方法
    lp = GetProvidersByPriv_name("");              //调用查询所有的方法
    dgv_Providers.DataSource = lp;
}
// 隐藏控件
public void Conceal()
{
     txt Priv adr.Enabled = false;
     txt Priv linkMan.Enabled = false;
     txt Priv name.Enabled = false;
     txt Priv tel.Enabled = false;
     tsb Btn Save.Enabled = false;
}
/// <summary>
/// 使用供货商名称查找供货商信息
/// </summary>
/// <param name="Priv name">供货商名称</param>
/// <returns>供货商表集合</returns>
public List<Providers> GetProvidersByPriv_name(string Priv_name)
{
    List<Providers> lp = new List<Providers>();
    SqlConnection con = new SqlConnection(conStr);    //创建数据库连接对象
    try
    {
       con.Open();                                    //打开连接
       string sql = "select * from providers where Privname like '%'+
```

```
        @Priv name+'%'";
        SqlCommand com = new SqlCommand(sql, con);      //创建命令执行对象
        //添加参数
        com.Parameters.Add("@Priv name", SqlDbType.VarChar, 50).Value =
        Priv_name;
        SqlDataReader dr = com.ExecuteReader();         //获取数据读取对象
        while (dr.Read())
        {
            Providers p = new Providers();
            p.Priv id = Convert.ToInt32(dr["Privid"]);
            p.Priv name = dr["Privname"].ToString();
            p.Priv_linkMan = dr["PrivlinkMan"].ToString();
            p.Priv_tel = dr["Privtel"].ToString();
            p.Priv_adr = dr["Privadr"].ToString();
            lp.Add(p);
        }
        dr.Close();                                     //关闭数据读取对象
    }
    catch (Exception)
    {
        MessageBox.Show("系统出现异常，请您稍后再试！");
    }
    finally
    {
        con.Close();                                    //关闭数据库连接
    }
    return lp;
}
```

根据厂商名称查询，代码如示例代码 19-12 所示，

示例代码 19-12

```
private void tsb Btn Lookup Click(object sender, EventArgs e)
{
    lp = GetProvidersByPriv name(tsb Txt Lookup.Text);
                                        //调用查询供应商详细信息的方法
    dgv_Providers.DataSource = lp;
}
```

2. 修改或添加供应商信息

先在供应商信息列表中选中要修改的项，然后单击“修改”图片按钮，添加、删除和查询按钮变为禁用，保存按钮启用。与此同时系统会自动将选中的供应商信息显示到下面供应商基本信息文本框中。操作员对文本框中的供应商信息进行修改，之后单击“保存”图片按钮，系统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入之后会弹出相应的提示对话框。

单击“添加”图片按钮，文本输入启用，修改、删除和查询按钮变为禁用。操作人员可以输入供应商的信息，单击“保存”按钮，系统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入之后会弹出相应的提示对话框。

注意： 厂家名称和地址是必填的，并且厂家名称不能重复。

其具体实现代码如示例代码 19-13 所示。

示例代码 19-13

```
//修改事件
private void tsb_Btn_Amend_Click(object sender, EventArgs e)
{
    type = 0;
    Display();                                  //调用显示控件的方法
    tsb_Btn_Add.Enabled = false;                //将添加按钮置为禁用
    tsb_Btn_Delete.Enabled = false;
    tsb_Txt_Lookup.Enabled = false;
    tsb_Btn_Lookup.Enabled = false;
    txt_Priv_name.Text = dgv_Providers.CurrentRow.Cells[1].Value. ToStr-
    ing();
    txt_Priv_adr.Text = dgv_Providers.CurrentRow.Cells[4].Value.ToStr-
    ing();
    txt_Priv_linkMan.Text = dgv_Providers.CurrentRow.Cells[2].Value.To-
    String();
    txt_Priv_tel.Text = dgv_Providers.CurrentRow.Cells[3].Value.ToStr-
    ing();
    txt_Priv_name.Focus();
}
//保存事件
private void tsb_Btn_Save_Click(object sender, EventArgs e)
{
    if (vailInput())
    {
        if (type == 0)                          //当保存的是修改操作
        {
            string sql =
                    string.Format("update providers set PrivName='{0}',"
                    +"PrivLinkMan='{1}',PrivTel='{2}',PrivAdr='{3}' "
                    +"where PrivId={4}", txt_Priv_name.Text,
                    txt_Priv_linkMan.Text, txt_Priv_tel.Text, txt_Priv_
                    adr. Text, Convert.ToInt32(dgv_Providers.CurrentRow.
                    Cells[0].Value));
            int result = GetProvidersChangeBySql(sql);
                                                //调用方法执行修改操作
            if (result > 0)
            {
                MessageBox.Show("修改成功! ", "提示");
            }
        }
        else if (type == 1)                     //当保存的是添加操作时执行
        {
            string sql = string.Format ("insert into providers (PrivName,
            PrivLinkMan, PrivTel, PrivAdr)" +"values('{0}', '{1}','{2}',
            '{3}')", txt_Priv_name.Text, txt_Priv_linkMan.Text,txt_Priv_
            tel.Text, txt_Priv_adr.Text);
            foreach (Providers item in lp)  //判断该厂家信息是否已经存在
            {
                if (item.Priv_name == txt_Priv_name.Text)
                {
                    MessageBox.Show("你要添加的信息已存在! ", "提示");
                    return;
                }
            }
            int result = GetProvidersChangeBySql(sql);
                                                //调用方法执行添加操作
            if (result > 0)
```

```
            {
                MessageBox.Show("添加成功！", "提示");
            }
        }
        lp = GetProvidersByPriv_name("");   //查询所有供应商信息
        dgv_Providers.DataSource = lp;      //将保存后的结果绑定到展示空间
        Cancel();
        Clear();
    }
}
//非空验证
private bool vailInput()
{
    if (txt_Priv_adr.Text.Trim() == "")
    {
        MessageBox.Show("地址不能为空！","提示");
        return false;
    }
    if (txt_Priv_linkMan.Text.Trim() == "")
    {
        MessageBox.Show("联系人不能为空！", "提示");
        return false;
    }
    if (txt_Priv_name.Text.Trim() == "")
    {
        MessageBox.Show("厂家名称不能为空！", "提示");
        return false;
    }
    if (txt_Priv_tel.Text.Trim() == "")
    {
        MessageBox.Show("联系电话不能为空！", "提示");
        return false;
    }
        return true;
}
/// <summary>
/// 执行数据库增删改操作的方法
/// </summary>
/// <param name="sql">要执行的 sql 语句</param>
/// <returns>数据库受影响的行数</returns>
public int GetProvidersChangeBySql(string sql)
{
    int i = 0;
    SqlConnection con = new SqlConnection(conStr);//数据库连接对象
    try
    {
        con.Open();                                  //打开数据库连接
        SqlCommand com = new SqlCommand(sql, con);
        i = com.ExecuteNonQuery();                   //执行方法，得到受影响行数
    }
    catch (Exception)
    {
        MessageBox.Show("系统繁忙,请稍后再试!", "提示");
    }
    finally
    {
        con.Close();                                 //关闭数据库连接对象
    }
    return i;                                //返回执行 SQL 语句数据库受影响行数
```

```
}
```

3．删除供应商信息

在供应商信息列表中选中要删除的供应商信息项，单击删除按钮系统会弹出“你确定要删除此条信息吗？”的对话框。单击“确定”按钮，系统会提示删除成功或失败。一般涉及数据删除必须要给予客户提示，以免操作发生，实现的具体代码如示例代码 19-14 所示。

示例代码 19-14

```
//删除事件
private void tsb_Btn_Delete_Click(object sender, EventArgs e)
{
    string sql = " delete from providers where PrivId=" + Convert.ToInt32
     (dgv_Providers.CurrentRow.Cells[0].Value);
    DialogResult dr = MessageBox.Show("你确定要删除此信息吗？", "提示",
    MessageBoxButtons.OKCancel);
    if (dr == DialogResult.OK)
    {
        int result = GetProvidersChangeBySql(sql); //调用方法执行删除操作
        if (result > 0)
        {
            MessageBox.Show("删除成功！", "提示");
            lp = GetProvidersByPriv_name("");   //调用查询供应商所有信息的方法
            dgv_Providers.DataSource = lp;      //重新绑定显示控件
        }
        else
        {
            MessageBox.Show("此信息与商品信息有关联请先删除商品信息！", "提示");
        }
    }
 }
```

注意：该段代码里用到的方法的具体实现过程前面已阐述过，这里不再重复。

19.5.7　品牌资料维护功能

1．查询品牌信息

在品牌资料维护窗体加载时，保存按钮和文本框处于禁用状态，与此同时系统会将所有的品牌信息查询出来并显示。为方便用户查找特定品牌的详细信息，程序提供了根据品牌名称查询品牌详细信息的方法。其具体的实现代码如示例代码 19-15 所示。

示例代码 19-15

```
//窗体加载
private void Frm_CardsInfo_Load(object sender, EventArgs e)
{
    lc = GetCardsByCards_name("");          //调用查询所有品牌信息的方法
```

```
        dgv Cards.DataSource = lc;
        Conceal();//设置控件为禁用状态
    }
    /// <summary>
    /// 使用品牌名称查询商品信息
    /// </summary>
    /// <param name="Cards name">品牌名称</param>
    /// <returns>品牌表集合</returns>
    public List<CardsInfo> GetCardsByCards_name(string Cards_name)
    {
        List<CardsInfo> lc = new List<CardsInfo>();
                                        //存储查询返回的商品信息的集合
        SqlConnection con = new SqlConnection(conStr);    //创建数据库连接对象
        try
        {
            con.Open();                                   //打开连接
            string sql = string.Format("select * from Cards where Cardsname like
            '%{0}%'", Cards name);
            SqlCommand com = new SqlCommand(sql, con);
            SqlDataReader dr = com.ExecuteReader();
            while (dr.Read())
            {
                CardsInfo u = new CardsInfo();
                u.CardsId = Convert.ToInt32(dr["Cardsid"]);
                u.CardsName = dr["Cardsname"].ToString();
                lc.Add(u);
            }
            dr.Close();//关闭 SqlDataReader 对象
        }
        catch (Exception)
        {
            MessageBox.Show("系统出现异常，请您稍后再试！");
        }
        finally
        {
            con.Close();                            //关闭数据库连接对象
        }
        return lc;                                  //返回查询返回的结果
    }
    /// <summary>
    /// 隐藏控件
    /// </summary>
    public void Conceal()
    {
        txt Cards Name.Enabled = false;             //设置控件状态为禁用
        tsb Btn Save.Enabled = false;
    }
```

2．修改或添加供应商信息

先在品牌信息列表中选中要修改的项，然后单击修改图片按钮，添加、删除和查询按钮将变为禁用，保存按钮将启用。与此同时，系统会自动将选中的品牌信息显示到下面的品牌基本信息文本框中。操作员对文本框中的品牌信息进行修改，之后单击保存图片按钮，系统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入之后会弹出相应的提示对话框。

单击“添加”图片按钮，文本输入启用，修改、删除和查询按钮变为禁用。操作人员可以输入品牌的信息，单击“保存”按钮，系统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入之后会弹出相应的提示对话框。

注意：品牌编号自动产生无法修改，品牌名称要使用全称。

其具体实现代码如示例代码 19-16 所示。

示例代码 19-16

```
//添加事件
private void tsb_Btn_Add_Click(object sender, EventArgs e)
{
    type = 1;   //修改 type 的值，为方便保存操作时判断是修改还是添加操作
    Display();
    tsb Btn Amend.Enabled = false;
    tsb Txt Lookup.Enabled = false;
    tsb Btn Delete.Enabled = false;
    tsb Btn Lookup.Enabled = false;
    txt Cards Name.Focus();
}
//修改事件
private void tsb Btn Amend Click(object sender, EventArgs e)
{
    type = 0;   //修改 type 的值，为方便保存操作时判是修改还是添加操作
    Display();
    tsb Btn Add.Enabled = false;
    tsb_Btn_Delete.Enabled = false;
    tsb_Txt_Lookup.Enabled = false;
    tsb_Btn_Lookup.Enabled = false;
    txt_Cards_Id.Text = dgv_Cards.CurrentRow.Cells[0].Value.ToString();
    txt_Cards_Name.Text = dgv_Cards.CurrentRow.Cells[1].Value.ToString();
    txt_Cards_Name.Focus();
}
//保存
private void tsb Btn Save Click(object sender, EventArgs e)
{
    if(txt Cards Name.Text.Trim()==""|| string.IsNullOrEmpty(txt Cards
    Name.Text.Trim()))
    {
       MessageBox.Show("商品名称不能为空！");
       return;
    }
    if (type == 0)//type 为 0 时，执行修改操作
    {
       string sql =string.Format("update cards set cardsname='{0}' where
       CardsId={1}", txt Cards Name.Text, Convert.ToInt32(txt Cards Id.
       Text));
       int result = executeQuery(sql);
       if (result > 0)
       {
          MessageBox.Show("修改成功！");
       }
       else
       {
          MessageBox.Show("修改失败！");
       }
    }
```

```
    else if (type == 1)          //type 为 1 时，执行添加操作
    {
        foreach (CardsInfo item in lc)
                              //添加之前首先判断该品牌信息是否已经存在，存在则提示操作
        {
            if (item.CardsName == txt Cards Name.Text)
            {
                MessageBox.Show("该商品品牌已存在","提示");
                return;
            }
        }
        string sql = string.Format("insert into dbo.cards(cardsName)values
        ('{0}');",txt_Cards_Name.Text);
        int result = executeQuery(sql);
        if (result > 0)
        {
            MessageBox.Show("添加成功！");
        }
        else
        {
            MessageBox.Show("添加失败！");
        }
    }
    lc = GetCardsByCards_name("");      //调用查询所有品牌信息的方法
    dgv Cards.DataSource = lc;
    Cancel();
}
//执行增删改操作的方法
private int executeQuery(string sql)
{
    SqlConnection con = new SqlConnection(conStr);//连接数据库对象
    int resule = 0;                                //执行 SQL 语句返回的结果
    try
    {
        con.Open();                                //打开连接
        SqlCommand cmd = new SqlCommand(sql, con);
        resule = cmd.ExecuteNonQuery();            //执行增删改
    }
    catch (Exception)
    {
        MessageBox.Show("此信息与商品信息有关联请先删除商品信息！", "提示");
    }
    finally
    {
        con.Close();                               //关闭连接
    }
        return resule;                             //返回结果
}
```

3．删除品牌信息

在品牌信息列表中选中要删除的品牌信息项，单击“删除”按钮时系统会弹出“你确定要删除此条信息吗？”的对话框。单击“确定”按钮，系统会提示删除成功或失败。实现的具体代码如示例代码 19-17 所示。

示例代码 19-17

```
//删除按钮事件
private void tsb_Btn_Delete_Click(object sender, EventArgs e)
{
    //判断当有选择项的时候
    if (dgv_Cards.SelectedRows.Count > 0)
    {
        if (MessageBox.Show("确定删除该条信息？", "提示",
        MessageBoxButtons.OKCancel) == DialogResult.OK)
        {
            string sql = string.Format("delete from cards where CardsId={0}",
            Convert.ToInt32 (dgv Cards.SelectedRows[0].Cells[0]. Value));
            int result = executeQuery(sql);
            if (result > 0)
            {
                MessageBox.Show("删除成功！", "提示");
                lc = GetCardsByCards_name("");   //调用查询所有品牌信息的方法
                dgv Cards.DataSource = lc;
            }
        }
    }
}
```

19.5.8　操作员资料维护功能

1. 查询用户信息

在用户资料维护窗体加载时，保存按钮、用户类型单选按钮和文本框处于禁用状态，与此同时系统会将所有的用户信息查询出来并显示。为方便管理员查找用户的详细信息，程序提供了根据用户名查询用户详细信息的方法。其具体的实现代码如示例代码 19-18 所示。

示例代码 19-18

```
//窗体加载
private void Frm UserInfo Load(object sender, EventArgs e)
{
    tsb Btn Save.Enabled = false;
    //调用查询所有用户信息的方法，绑定信息到显示控件
    dgv_UserInfo.DataSource = GetUserInfoByUserloginIdOrUserName("");
}
//根据条件查询用户信息
private void tsb_Btn_Lookup_Click(object sender, EventArgs e)
{
    if (tsb_Txt_Lookup.Text.Trim() == "")      //判断查询条件不为空
    {
        MessageBox.Show("请输入查询条件！", "提示", MessageBoxButtons.OK,
        MessageBoxIcon.Asterisk);
    }
    else
    {
```

```
        lu=GetUserInfoByUserloginIdOrUserName(tsb Txt Lookup.Text);
                                                //根据条件查询
        dgv_UserInfo.DataSource = lu;           //将新查询的结果绑定到显示控件
    }
}
/// <summary>
/// 使用用户登录名或是真实姓名查询所有 UserInfo 表信息
/// </summary>
/// <param name="UserloginIdOrUserName">用户登录名或是真实姓名</param>
/// <returns>UserInfo 集合</returns>
public List<UserInfo> GetUserInfoByUserloginIdOrUserName(string Userlo-
ginIdOrUserName)
{
    lu = new List<UserInfo>();
    SqlConnection con = new SqlConnection(DBHelper.conStr);
    try
    {
        con.Open();                             //打开数据库连接
        //查询所有非超级管理员
        string sql = string.Format("select * from userInfo where userLoginId
        like '%{0}%'and userRole!=0 or userName like '%{0}%' and userRole!=0",
        UserloginIdOrUserName);
        SqlCommand com = new SqlCommand(sql, con);
        SqlDataReader dr = com.ExecuteReader();
        while (dr.Read())
        {
            UserInfo u = new UserInfo();
            u.UserId = Convert.ToInt32(dr["UserId"]);
            //由于篇幅的限制，详细代码参考光盘源文件
            …
            u.UserType = dr["UserRole"].ToString() == "1" ? "销售员" : "采
            购员";
            lu.Add(u);
        }
        dr.Close();                             //关闭 SqlDataReader 对象
    }
    catch (Exception)
    {
        MessageBox.Show("系统出现异常，请您稍后再试！", "系统提示", Message-
        BoxButtons.OK);
    }
    finally
    {
        con.Close();                            //关闭数据库连接对象
    }
    return lu;
}
```

2．修改或添加用户信息

先在用户信息列表中选中要修改的项，然后单击“修改”图片按钮，添加、删除和查询按钮变为禁用，保存按钮启用。与此同时系统会自动将选中的用户信息显示到下面用户基本信息文本框中。管理员对文本框中的用户信息进行修改，之后单击保存图片按钮，系

统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入，之后会弹出相应的提示对话框。

单击“添加”图片按钮，文本输入启用，修改、删除和查询按钮变为禁用。管理员可以输入要添加的用户信息，单击“保存”按钮，系统首先会判断要执行的是添加操作还是修改操作。在确定操作后，系统会检测操作员的数据输入，之后会弹出相应的提示对话框。

注意：登录账号注册后不能再修改。

其具体实现代码如示例代码 19-19 所示。

示例代码 19-19

```
//修改事件
private void tsb_Btn_Amend_Click(object sender, EventArgs e)
{
    type = 0;
    //判断是否已选中要操作的对象
    if (dgv_UserInfo.SelectedRows.Count > 0)
    {
        //将选中的信息显示到文本框中
        tsb_Btn_Save.Enabled = true;
        tsb_Btn_Add.Enabled = false;
        tsb_Btn_Delete.Enabled = false;
        pnl_baseInfo.Enabled = true;
        txt_UserloginId.Enabled = false;
        txt_UserloginId.Text = dgv_UserInfo.SelectedRows[0].Cells[1].
        Value. ToString();
        txt_UserName.Text = dgv_UserInfo.SelectedRows[0].Cells[2].Value.
        ToString();
        txt_UserPass.Text = dgv_UserInfo.SelectedRows[0].Cells[3].Value.
        ToString();
        if (dgv_UserInfo.SelectedRows[0].Cells["UserRole"].Value. ToStr-
        ing() == "2")rdo_sellUser.Checked=true;
        else if (dgv_UserInfo.SelectedRows[0].Cells["UserRole"].Value.
        ToString() == "1")rdo_buyUser.Checked=true;
    }
}
//添加事件
private void tsb_Btn_Add_Click(object sender, EventArgs e)
{
    type = 1;
    tsb_Btn_Save.Enabled = true;              //启用控件
    tsb_Btn_Amend.Enabled = false;            //禁用控件
    tsb_Btn_Delete.Enabled = false;
    pnl_baseInfo.Enabled = true;
    txt_UserloginId.Enabled = true;
    txt_clear();
}
//非空验证
public bool Vail()
{
    if (txt_UserloginId.Text.Trim() == "" || String.IsNullOrEmpty(txt_Use-
    rloginId.Text))
    {
        MessageBox.Show("请输入用户名!","提示",MessageBoxButtons.OK,Message
        BoxIcon.Asterisk);
        return false;
```

```
    }
    else if (txt UserName.Text.Trim() == "" || String.IsNullOrEmpty(txt
    UserName.Text))
    {
        MessageBox.Show("请输入真实姓名！", "提示", MessageBoxButtons.OK,
        MessageBoxIcon.Asterisk);
        return false;
    }
    else if (txt UserPass.Text.Trim() == "" || String.IsNullOrEmpty(txt
    UserPass.Text))
    {
        MessageBox.Show("请输入密码！","提示",MessageBoxButtons.OK,Message
        BoxIcon.Asterisk);
        return false;
    }
    else if (!txt UserPass1.Text.Trim().Equals(txt UserPass. Text.Trim
    ()))
    {
        MessageBox.Show("两次密码不一致！", "提示", MessageBoxButtons.OK,
        MessageBoxIcon.Asterisk);
        return false;
    }
    else
    {
        return true;
    }
}
//保存按钮事件
private void tsb_Btn_Save_Click(object sender, EventArgs e)
{
    if (type == 0)                                  //修改操作
    {
        if (Vail())                                 //非空验证通过后
        {
            int userRole = rdo buyUser.Checked == true ? 2 : 1;
            string sql = string.Format("update userInfo set UserName=
            '{0}',UserPass='{1}',userrole={3} where UserloginId='{2}'",
            txt UserName.Text, txt UserPass.Text, txt UserloginId.Text,
            userRole);
            int result = GetUserInfoChangeBySql(sql);
            if (result > 0)
            {
                MessageBox.Show("修改成功！", "提示", MessageBoxButtons.OK,
                MessageBoxIcon.Asterisk);
                tsb Btn Delete.Enabled = true;
            }
        }
    }
    else if (type == 1)                             //添加操作
    {
        if (Vail())                                 //非空验证通过后
        {
            int userRole = rdo buyUser.Checked == true ? 2 : 1;
            string sql =
                  string.Format("insert into dbo.userInfo(UserloginId,
                  UserName, UserPass,UserRole)values('{0}','{1}', '{2}',
                  {3})",
                  txt_UserloginId.Text, txt_UserName.Text, txt_UserPass.
```

```
                    Text, userRole);
            foreach (UserInfo item in lu)       //判断用户名是否已经用过
            {
                if (item.UserloginId.Equals(txt_UserloginId.Text))
                {
                    MessageBox.Show("该用户已存在!请修改登录名", "提示", Message
                    BoxButtons.OK, MessageBoxIcon.Asterisk);
                    return;
                }
            }
            int result = GetUserInfoChangeBySql(sql);   //调用方法，执行操作
            if (result > 0)
            {
                MessageBox.Show("添加成功! ", "提示", MessageBoxButtons.OK,
                MessageBoxIcon.Asterisk);
            }
        }
    }
    lu = GetUserInfoByUserloginIdOrUserName("");
                                            //调用方法，重新查询用户信息
    txt_clear();
    pnl_baseInfo.Enabled = false;
    dgv_UserInfo.DataSource = lu;         //将新查询的结果绑定到显示控件
}
/// <summary>
/// 用 SQL 语句进行增加修改查操作
/// </summary>
/// <param name="sql">sql 语句</param>
public int GetUserInfoChangeBySql(string sql)
{
    int i = 0;
    SqlConnection con = new SqlConnection(DBHelper.conStr);
    try
    {
        con.Open();                           //打开数据库连接
        SqlCommand com = new SqlCommand(sql, con);
        i = com.ExecuteNonQuery();
    }
    catch (Exception)
    {
        MessageBox.Show("系统出现异常，请您稍后再试! ", "系统提示", Message-
        BoxButtons.OK);
    }
    finally
    {
        con.Close();                          //关闭数据库连接
    }
    return i;
}
```

3．删除用户信息

在以后信息列表中选中要删除的用户信息项，单击“删除”按钮系统会弹出“你确定要删除此条信息吗？”的对话框。单击“确定”按钮，系统会提示删除成功或失败。实现的具体代码如示例代码 19-20 所示。

示例代码 19-20

```
//删除事件
private void tsb Btn Delete Click(object sender, EventArgs e)
{
    if (dgv UserInfo.SelectedRows.Count > 0)
    {
        //获取要删除的操作员 ID，注意在试用前先转为 Int 类型
        int userId =Convert.ToInt32(dgv UserInfo.SelectedRows[0]. Cells-
        [0].Value);
        string sql = "delete from userInfo where UserId="+userId;
        int result = GetUserInfoChangeBySql(sql);
        if (result > 0)
        {
            MessageBox.Show("删除成功！", "提示", MessageBoxButtons.OK,
            MessageBoxIcon.Asterisk);
            lu = GetUserInfoByUserloginIdOrUserName("");
            dgv_UserInfo.DataSource = lu;
            txt_clear();
            pnl_baseInfo.Enabled = false;
        }
    }
}
```

19.5.9　销售出库功能

提供销售商品。当光标进入商品名称文本框时，按下 Enter 键，选择要销售的商品。按下 Enter 键，输入销售数量，按下 Enter 键。之后单击“保存”图片按钮，提示销售成功，否则提示销售失败！（销售提示：当销售的商品数量小于该商品的库存警报数量时，提示该商品已经到达销售警报位置，需要进货）。

1. 查询所有商品信息

当光标进入商品名称文本框时，按下 Enter 键后系统将弹出库存查询窗体，与此同时将该窗体上的修改、删除、取消和查询图片按钮设为隐藏，并显示所有的商品信息列表。为了方便用户的操作，系统提供了按商品名称（支持模糊查询）查询商品的详细信息。具体实现代码如示例代码 19-21 所示。

示例代码 19-21

```
private static readonly string conStr = DBHelper.conStr;
                                                    //数据库连接字符串
List<StoreRoom> ls = new List<StoreRoom>();     //存储库存信息的集合
//窗体加载
private void Frm StoreQuery Load(object sender, EventArgs e)
{
    dgv Store.AutoGenerateColumns = false;
    ls = GetStoreRoomByProt Name(common.Prot name);
    dgv_Store.DataSource = ls;
    if (common.Prot_Enter 键 AndOut)
    {
        tsb Btn Amend.Visible = false;
        tsb_Btn_Cancel.Visible = false;
```

```
        tsb Btn Delete.Visible = false;
        tsb Btn Lookup.Visible = false;
        tsb Txt Lookup.Visible = false;
        common.Prot_Enter键AndOut = false;
    }
    common.Prot_name = "";
}
//商品名按键事件
private void txt Prot Name KeyDown(object sender, KeyEventArgs e)
{
    try
    {
        if (e.KeyCode == Keys.Enter键)
        {
            common.Prot_name = txt_Prot_Name.Text;
            Frm_StoreQuery fs = new Frm_StoreQuery();
            common.Prot_Enter键AndOut = true;
            fs.ShowDialog();
            StoreRoom s = common.s;
            txt Prot Bigunit.Text = s.Prot Bigunit;
            txt Prot Name.Text = s.Prot name;
            txt Prot Rate.Text = s.Prot rate.ToString();
            txt Prot Retailprice.Text = s.Prot retailprice.ToString();
            txt Prot Smallunit.Text = s.Prot smallunit;
            txt Prot Tradeprice.Text = s.Prot tradeprice.ToString();
            cbb Cards Name.Text = s.Cards name;
            cbb_Priv_Name.Text = s.Priv_name;
            dtp_Prot_Date.Value = s.Prot_date;
            lb_Prot_id.Text = s.Prot_id.ToString();
            txt_Out_Num.Focus();
        }
    }
    catch (Exception)
    {
        MessageBox.Show("请选择一项");
    }
}
/// <summary>
/// 使用商品名称查询库存信息
/// </summary>
/// <param name="Prot_Name">商品名称</param>
public List<StoreRoom> GetStoreRoomByProt Name(string Prot Name)
{
    List<StoreRoom> ls = new List<StoreRoom>();
    SqlConnection con = new SqlConnection(conStr);
    try
    {
        con.Open();                              //打开数据库连接
        string sql = string.Format("select productInfo.*,Cards.Cardsname,"
             +"providers.Privname,storeRoom.storelastNum,storeRoom. Sto-
             resum," +"storeRoom.storeid  from productInfo,Cards,provid-
             ers, storeRoom "+"where productInfo.ProtCard=Cards.Cardsid
             and "+"productInfo.ProtproviderId=providers.Privid and "+
             "productInfo.Protid=storeRoom.Protid and "+"productInfo.
             Protname like '%{0}%'", Prot Name);
             SqlCommand com = new SqlCommand(sql, con);
             SqlDataReader dr = com.ExecuteReader();
        while (dr.Read())
        {
            StoreRoom sr = new StoreRoom();
```

```
            //由于篇幅的限制，此处同上一行类似代码省略
            …
            CardsInfo cardsInfo = new CardsInfo();
            cardsInfo.CardsId = Convert.ToInt32(dr["ProtCard"]);
            Providers providers = new Providers();
            providers.Priv_id = Convert.ToInt32(dr["ProtproviderId"]);
            sr.Prot_date = Convert.ToDateTime(dr["Protdate"]);
            //由于篇幅的限制，此处同上一行类似代码省略
            …
            ls.Add(sr);
        }
        dr.Close();                         //关闭 SqlDataReader 数据读取对象
    }
    catch (Exception)
    {
        MessageBox.Show("系统出现异常，请你稍后再试！");
    }
    finally
    {
        con.Close();                        //关闭数据库连接
    }
    return ls;
}
```

2．绑定品牌信息和供应商信息

在销售出库窗体加载时，查询出品牌信息和供应商信息并将信息绑定到下拉列表框中，以备客户选择。其实现代码具体如示例代码 19-22 所示。

示例代码 19-22

```
//窗体加载
private void Frm_SellOut_Load(object sender, EventArgs e)
{
    dgv_Store.AutoGenerateColumns = false;          //只显示编辑的列
    cbb_Cards_Name.DisplayMember = "Cardsname";     //品牌信息显式值
    cbb_Cards_Name.ValueMember = "Cardsid";         //品牌信息隐式值
    cbb_Cards_Name.DataSource = LoginInfo.GetCardsByCards_name("");
    cbb_Priv_Name.DisplayMember = "Priv_name";      //厂家信息显式值
    cbb_Priv_Name.ValueMember = "Priv_id";          //厂家信息隐式值
    cbb_Priv_Name.DataSource = LoginInfo.GetProvidersByPriv_name("");
}
public class LoginInfo
{
    public static UserInfo LoginUserInfo;
    //连接数据库字符串
    private static radonly string conStr =DBHelper.conStr;
    /// <summary>
    /// 使用品牌名称查询商品信息
    /// </summary>
    /// <param name="Cards_name">品牌名称</param>
    /// <returns>品牌表集合</returns>
    public static List<CardsInfo> GetCardsByCards_name(string Cards_name)
    {
```

```
        List<CardsInfo> lc = new List<CardsInfo>();
        SqlConnection con = new SqlConnection(conStr);
        con.Open();
        string sql = "select * from Cards where Cardsname like '%'+@Car-
        dsname+'%'";
        SqlCommand com = new SqlCommand(sql, con);
        com.Parameters.Add("@Cardsname", SqlDbType.VarChar, 50).Value =
        Cards name;
        SqlDataReader dr = com.ExecuteReader();
        while (dr.Read())
        {
            CardsInfo cardsInfo = new CardsInfo();
            cardsInfo.CardsId = Convert.ToInt32(dr["Cardsid"]);
            cardsInfo.CardsName = dr["Cardsname"].ToString();
            lc.Add(cardsInfo);
        }
        dr.Close();
        con.Close();
        return lc;
    }
    /// <summary>
    /// 使用供货商名称查找供货商信息
    /// </summary>
    /// <param name="Priv name">供货商名称</param>
    /// <returns>供货商表集合</returns>
   public static List<Providers> GetProvidersByPriv name(string Privname)
   {
       List<Providers> lp = new List<Providers>();
       SqlConnection con = new SqlConnection(conStr);
       con.Open();
       string sql = "select * from providers where Privname like '%'+Pri-
       vname+'%'";
       SqlCommand com = new SqlCommand(sql, con);
       SqlDataReader dr = com.ExecuteReader();
       while (dr.Read())
       {
           Providers p = new Providers();
           p.Priv id = Convert.ToInt32(dr["Privid"]);
           p.Priv name = dr["Privname"].ToString();
           p.Priv linkMan = dr["PrivlinkMan"].ToString();
           p.Priv tel = dr["Privtel"].ToString();
           p.Priv adr = dr["Privadr"].ToString();
           lp.Add(p);
       }
       dr.Close();
       con.Close();
       return lp;
   }
}
```

3．实现销售功能

在销售数量文本框中输入数量，按下 Enter 键，将要销售的商品信息添加到下面的商品信息列表中，单击保存按钮系统弹出提示对话框“销售成功或失败”。具体实现代码如示例代码 19-23 所示。

示例代码 19-23

```
//保存事件
private void tsb Btn Save Click(object sender, EventArgs e)
{
    //循环将销售出库记录保存到数据库
    try
    {
        if (dgv Store.DataSource != null)
        {
            int i = GetOrersChangeBySql(DateTime.Now, LoginInfo.LoginUser
            Info. UserId, money, sum, 2);
            if (i > 0)
            {
                foreach (StoreRoom sr in ls)
                {
                    //调用添加新的订单明细表方法，将销售信息添加到订单表中
                    GetOrerDetailsChangeBySql(i, sr.Prot_id, sr.In_Out_
                    StoreRoom);
                    string sql = "update storeRoom set storesum=@store_sum
                    where Protid=@Prot_id";
                    int store_sum = sr.Store_sum - sr.In_Out_StoreRoom;
                    if (GetStoreRoomChangeBySql(sql, sr.Prot id, store sum,
                    sr.Store lastNum) > 0)
                        MessageBox.Show("商品“" + sr.Prot name + "”数量“" +
                        sr. In_Out_StoreRoom + "”销售成功");
                    else
                        MessageBox.Show("商品“" + sr.Prot_name + "”数量“" +
                        sr.In Out StoreRoom + "”销售失败");
                }
                ls.Clear();
                Clear();
                dgv_Store.DataSource = null;
                num = 0;
                sum = 0;
                money = 0;
            }
            else
            {
                MessageBox.Show("订单总表添加失败，请重新再试");
            }
        }
        else
            MessageBox.Show("请添加商品信息");
    }
    catch (Exception)
    {
        MessageBox.Show("失败");
    }
}
//销售数量按键事件
private void txt_Out_Num_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter 键)
    {
        try
        {
            StoreRoom s = common.s;
            if (Convert.ToInt32(txt_Out_Num.Text) > s.Store_sum)
```

```
                                                    //当销售数量大于库存时
            {
                MessageBox.Show("库存不够，请进货，或减少销售");
                txt Out Num.SelectAll();
                return;
            }
            if (Convert.ToInt32(txt Out Num.Text) <= 0)
                                                    //销售数量不能为负数
            {
                MessageBox.Show("销售出库不能为负值和");
                txt Out Num.SelectAll();
                return;
            }
            else
            {
                bool b = true;
                foreach (StoreRoom var in ls)
                {
                    if (var.Prot id.ToString() == lb Prot id.Text)
                                                    //判断该信息是否存在
                    {
                        //当销售数量大于库存时
                        if (var.In Out StoreRoom + Convert.ToInt32(txt Out
                        Num.Text) > var.Store sum)
                        {
                            MessageBox.Show("库存不够，请进货，或减少销售");
                            txt_Out_Num.SelectAll();
                            return;
                        }
                        var.In_Out_StoreRoom += Convert.ToInt32(txt_Out_Num.
                        Text);
                        sum += Convert.ToInt32(txt_Out_Num.Text);
                        money += Convert.ToInt32(txt_Out_Num.Text) * var.Prot_
                        tradeprice;
                        //当商品库存数量小于该商品的库存警报时
                        if (var.Store sum - var.In Out StoreRoom < var.Store
                        lastNum)
                        {
                            MessageBox.Show("请注意，该商品销售后，该商品库存小于库存
                            下线，请尽快进货", "警告", MessageBoxButtons.OK, Messa-
                            geBoxIcon. Exclamation);
                        }
                        b = false;
                        break;
                    }
                }
                if (b)
                {
                    s.In_Out_StoreRoom = Convert.ToInt32(txt_Out_Num.Text);
                    num++;
                    sum += s.In_Out_StoreRoom;
                    money+= s.In_Out_StoreRoom * s.Prot_tradeprice;
                    //当商品库存数量小于该商品的库存警报时
                    if (s.Store sum - s.In Out StoreRoom < s.Store lastNum)
                    {
                        MessageBox.Show("请注意，该商品销售后，该商品库存小于库存下线，
                        请尽快进货", "警告", MessageBoxButtons.OK, MessageBoxIcon.
                        Exclamation);
                    }
```

```
                ls.Add(s);
            }
            common.s = null;
            dgv Store.DataSource = null;
            dgv Store.DataSource = ls;
            Clear();
            txt Prot Name.Focus();
            lb money.Text = money.ToString();
            lb num.Text = num.ToString();
            lb sum.Text = sum.ToString();
        }
    }
    catch (Exception)                          //捕获错误信息
    {
        MessageBox.Show("请输入正确的数字格式");
        txt Out Num.SelectAll();
    }
  }
}
```

4. 实现记录销售信息的功能

业务是：插入订单表，同时插入订单明细表还要更新商品库存。这里需要注意，在插入订单明细表时需要订单总表的 ID，需要使用“@@identity”来获取。代码如示例代码 19-24 所示。

示例代码 19-24

```
对商品销售信息进行记录，其具体实现代码如下：
/// <summary>
/// 对订单总表进行添加
/// </summary>
/// <param name="order_time">订单时间</param>
/// <param name="userId">操作员 ID</param>
/// <param name="order sum money">该订单商品总价</param>
/// <param name="order_sum_total">该订单商品总数量</param>
/// <param name="order_type">该订单类型 0 进货 1 退货 2 销售 3 销售退货</param>
/// <returns>订单总表编号</returns>
public int GetOrersChangeBySql(DateTime order_time, int userId, float
order_sum_money,
int order_sum_total, int order_type)
{
    SqlConnection con = new SqlConnection(conStr);
    int i = 0;
    try
    {
        con.Open();                         //打开数据库连接
        string sql = "insert orders values(@Order time, @UserId, @Order sum
        money, @Order sum total, @Order type)";
        SqlCommand com = new SqlCommand(sql, con);
        com.Parameters.Add("@order time", SqlDbType.DateTime).Value =
        order time;
        com.Parameters.Add("@userId", SqlDbType.Int).Value = userId;
        com.Parameters.Add("@order_sum_money", SqlDbType.Money).Value =
        order_sum_money;
        com.Parameters.Add("@order_sum_total", SqlDbType.Int).Value =
        order_sum_total;
```

```
            com.Parameters.Add("@order type", SqlDbType.Int).Value = order
            type;
            i = com.ExecuteNonQuery();                  //执行操作
            if (i > 0)
            {
                com.CommandText = "select @@identity";
                i = Convert.ToInt32(com.ExecuteScalar());
            }
        }
        catch (Exception)
        {
            throw;
        }
        finally
        {
            con.Close();                                //关闭数据库连接
        }
        return i;
    }
    /// <summary>
    /// 添加新的订单明细表
    /// </summary>
    /// <param name="order id">订单总表 ID</param>
    /// <param name="prot_id">商品编号 ID</param>
    /// <param name="order_det_sum">商品数量</param>
    /// <returns></returns>
    public int GetOrerDetailsChangeBySql(int order id, int prot id, int
    order det sum)
    {
        SqlConnection con = new SqlConnection(conStr);
        int i = 0;
        try
        {
            con.Open();                                 //打开数据库连接
            string sql = "insert orderDetails values(@Order id, @Prot id,
            @order_det_sum)";
            SqlCommand com = new SqlCommand(sql, con);
            com.Parameters.Add("@order id", SqlDbType.Int).Value = order id;
            com.Parameters.Add("@prot id", SqlDbType.Int).Value = prot id;
            com.Parameters.Add("@order det sum", SqlDbType.Int).Value = order
            det sum;
            i = com.ExecuteNonQuery();
        }
        catch (Exception)
        {
            throw;
        }
        finally
        {
            con.Close();                                //关闭数据库连接
        }
        return i;
    }
    /// <summary>
    /// 用 SQL 语句对库存进行添加修改
    /// </summary>
    /// <param name="sql">sql 语句</param>
    /// <param name="Prot id">商品 ID 外键</param>
    /// <param name="store_sum">商品库存</param>
```

```
/// <param name="store lastNum">库存下限</param>
/// <returns>sql 语句影响的行数</returns>
public int GetStoreRoomChangeBySql(string sql, int Prot_id, int store_sum,
int store lastNum)
{
    SqlConnection con = new SqlConnection(conStr);
    int i = 0;
    try
    {
        con.Open();     //打开数据库连接
        SqlCommand com = new SqlCommand(sql, con);
        com.Parameters.Add("@Prot_id", SqlDbType.Int).Value = Prot_id;
        com.Parameters.Add("@store_sum", SqlDbType.Int).Value = store_sum;
        com.Parameters.Add("@store_lastNum", SqlDbType.Int).Value = store_
        lastNum;
        i = com.ExecuteNonQuery();
    }
    catch (Exception)
    {
        throw;
    }
    finally
    {
        con.Close();         //关闭数据库连接
    }
    return i;
}
```

5. 实现将添加到销售列表中的商品删除

在实现销售功能时，每次在销售数量文本框中输入数量，按下 Enter 键，将要销售的商品信息添加到下面的商品销售信息列表中。为方便用户的操作，再次按下 Enter 键光标又一次进入了商品名称文本框中，按下 Enter 键又重复上一次的操作。就这样可以一次性向商品销售信息列表中添加多条数据，最后当单击“保存”按钮后一次性提交。在这个操作过程中，用户很有可能开始要选购某种商品，后面又不想要了，此时系统为方便销售人员操作，程序提供了可以将已选购商品从销售商品列表中删除的操作。具体实现代码如示例代码 19-25 所示。

示例代码 19-25

```
//删除事件
private void tsb Btn Delete Click(object sender, EventArgs e)
{
    DialogResult dr = MessageBox.Show("是否确定删除该商品？", "删除提示",
    MessageBoxButtons.OKCancel, MessageBoxIcon.Error);
    if (dr == DialogResult.Cancel)
        return;
    else
    {
        if (dgv Store.SelectedCells.Count <= 0)
        {
            return;
        }
        int i = Convert.ToInt32(dgv Store.CurrentRow.Cells ["cl Prot id"]
        .Value);                                              //格式转换
```

```
        foreach (StoreRoom var in ls)
        {
            if (var.Prot id == i)
            {
                common.s = var;
                break;
            }
        }
        ls.Remove(common.s);                               //删除
        common.s = null;
        dgv_Store.DataSource = null;
        dgv Store.DataSource = ls;
    }
}
```

注意：实现本功能时用到的集合和其他方法在前面已介绍过，这里不再重复。

19.5.10　销售退货功能

提供销售退货功能。当光标进入商品名称文本输入框时，按下 Enter 键，选择要退的商品，按下 Enter 键，输入退货数量，按下 Enter 键。之后单击“保存”图片按钮，输入退货原因。之后系统会提示退货成功，否则提示退货失败。

1．查询所有商品信息

光标进入商品名称文本框时，按下 Enter 键后系统将弹出库存查询窗体，与此同时将该窗体上的修改、删除、取消和查询图片按钮设为隐藏，并显示所有的商品信息列表。为了方便用户的操作，系统提供了按商品名称（支持模糊查询）查询商品的详细信息。具体实现代码如示例代码 19-26 所示。

示例代码 19-26

```
//窗体加载
private void Frm SellEnter键 Load(object sender, EventArgs e)
{
    dgv_Store.AutoGenerateColumns = false;//设置为手工创建列
    cbb_Cards_Name.DisplayMember = "Cardsname";
    cbb_Cards_Name.ValueMember = "Cardsid";
    //调用查询所有商品信息的方法，并将结果集绑定到下拉列表框中
    cbb Cards Name.DataSource= LoginInfo.GetCardsByCards name("");
    cbb Priv Name.DisplayMember = "Priv name";
    cbb Priv Name.ValueMember = "Priv id";
    cbb Priv Name.DataSource= LoginInfo.GetProvidersByPriv name("");
}
```

注意：具体实现查询所有商品信息的方法在前面已介绍过，这里不再具体阐述。

2．实现退货功能

在退货数量文本框中输入数量，按下 Enter 键，先将要退的商品信息添加到下面的商品信息列表中，单击“保存”按钮系统弹出提示对话框“退货成功或失败”。与此同时系统会记录下退货信息以便日后查看。具体实现代码如示例代码 19-27 所示。

示例代码 19-27

```
//保存按钮事件
private void tsb Btn Save Click(object sender, EventArgs e)
{
     try
     {
        if (dgv_Store.DataSource != null)
        {
            int i = GetOrersChangeBySql(DateTime.Now,LoginInfo.LoginUser-
            Info. UserId, money, sum, 3);
            if (i > 0)
            {
                foreach (StoreRoom sr in ls)
                {
                     GetOrerDetailsChangeBySql(i, sr.Prot id, sr.In Out
                     StoreRoom);
                     string sql = "update storeRoom set storesum=@store sum
                     where Protid=@Prot id";
                     int store sum = sr.Store sum + sr.In Out StoreRoom;
                     if(GetStoreRoomChangeBySql(sql,sr.Prot id,store sum,
                     sr.Store lastNum) > 0)
                     MessageBox.Show("商品“" + sr.Prot_name + "”数量“" + sr.
                     In_Out_StoreRoom + "”销售退货成功");
                     else
                     MessageBox.Show("商品“" + sr.Prot name + "”数量“" + sr.
                     In_Out_StoreRoom + "”销售退货失败");
                }
                ls.Clear();
                Clear();
                dgv Store.DataSource = null;
                num = 0;
                sum = 0;
            money = 0;
            }
            else
            {
                MessageBox.Show("订单总表添加失败，请重新再试！");
            }
        }
        else
            MessageBox.Show("请添加商品信息！");
    }
    catch (Exception)
    {
        MessageBox.Show("退货失败！");
    }
}
```

注意： 实现本功能中用到的方法和实现销售出库功能中基本类似，在进行商品销售退货的同时系统会记录下退货信息，实现记录退货信息的方法与对商品销售出库信息进行记录的操作类似，这里不再进行详细阐述。

3．实现将添加到退货列表中的商品删除

在实现退货功能时，每次在退货数量文本框中输入数量，按下 Enter 键，将要退的商品信息添加到下面的商品退货信息列表中。为方便用户的操作，再次按下 Enter 键光标又

一次进入了商品名称文本框中，按下 Enter 键又重复上一次的操作。就这样可以一次性向商品退货信息列表中添加多条数据，最后当单击“保存”按钮后一次性提交。在这个操作过程中，用户很有可能开始要退某种商品，后面又不想退了，此时系统为方便销售人员操作，提供了可以将已退商品从退货商品列表中删除的操作。由于此操作与销售出库中将添加到销售列表中的商品删除的实现方法基本一样，所以这里不再讲述。

19.5.11　采购入库功能

实现此功能的操作和销售出库基本类似，因此在此处不再讲解已介绍过的知识点，要想知道详情，请看销售出库部分。下面开始讲解不一样的部分。

当将需要采购的商品放入采购商品信息列表中后，系统会自动进行计算总采购数量和总价钱，具体实现代码如示例代码 19-28 所示。

示例代码 19-28

```
//键按下事件
private void txt Enter键 Num KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter键)
    {
        try
        {
            if (Convert.ToInt32(txt Enter键 Num.Text) <= 0)
            {
                MessageBox.Show("销售退货不能为负值和");
                txt_Enter键_Num.SelectAll();
                return;
            }
            bool b = true;
            foreach (StoreRoom var in ls)
            {
                if (var.Prot id==common.s.Prot id)
                {
                    var.In_Out_StoreRoom += Convert.ToInt32(txt_Enter键
                    _Num.Text);
                    sum += Convert.ToInt32(txt Enter键 Num.Text);
                    money += Convert.ToInt32(txt_Enter键_Num.Text)* var.
                    Prot_tradeprice;
                    b = false;
                    break;
                }
             }
             StoreRoom s = common.s;
             if (b)
             {
                 s.In Out StoreRoom = Convert.ToInt32(txt Enter键 Num.
                 Text);
                 num++;
                 sum += s.In_Out_StoreRoom;
                 money+= s.In_Out_StoreRoom * s.Prot_tradeprice;
                 ls.Add(s);
             }
             common.s = null;
```

```
            dgv Store.DataSource = null;
            dgv Store.DataSource = ls;
            Clear();
            txt Prot Name.Focus();
            lb money.Text = money.ToString();
            lb num.Text = num.ToString();
            lb sum.Text = sum.ToString();
        }
        catch (Exception)
        {
            MessageBox.Show("请输入正确的数字格式");
            txt Enter键 Num.SelectAll();
        }
    }
}
```

注意：实现本功能涉及的方法在前面已介绍过，这里不再阐述。

19.5.12　采购退货功能

采购退货功能的实现思路和销售退货的实现思路一样，在此不做过多的讲解。

值得注意的是当商品库存量为 0 时，想要彻底地删除该商品，即以后查询库存时再也看不到已删除的商品。要想实现此效果，需要两步：首先要确保库存中该商品数量为 0，其次需要到基本信息维护菜单中选择商品信息维护选项，单击该选项系统会弹出维护商品信息窗体，在其上的商品信息列表中找到要彻底删除的商品，单击“删除”按钮。系统会提示删除成功或失败。若显示删除成功，至此才彻底删除了需要删除的商品。

19.5.13　库存查询功能

在前面讲解销售商品时，每次当光标进入商品名称文本框时，按下 Enter 键系统会弹出一个窗体，其实那就是实现的库存查询功能。鉴于此，库存查询功能在这里不做详细讲解。值得一提的是在库存查询中有一个修改按钮，在库存商品信息列表中选中一行，单击“修改”按钮可以修改选中商品的库存下限。也就是当库存中的商品数量小于等于多少时，系统会提示你：该商品的库存量已到库存下限，请您进行采购！其具体实现代码如示例代码 19-29 所示。

示例代码 19-29

```
private void tsb_Btn_Amend_Click(object sender, EventArgs e)
{
    Frm Store LastNum FsL = new Frm Store LastNum();
    FsL.ShowDialog();
string sql =
            string.Format("update storeRoom set storelastNum={0} wherePro-
            tid={1}",common.store_lastNum, Convert.ToInt32(dgv_Store.
            CurrentRow.Cells[0].Value));
    int result = executeQuery(sql);
    if (result > 0)
    {
```

```
        MessageBox.Show("修改成功", "提示");
    }
    else
    {
        MessageBox.Show("修改失败","提示");
    }
    ls = GetStoreRoomByProt_Name(common.Prot_name);
    dgv_Store.DataSource = ls;
}
//进行库存下限修改的方法
public int executeQuery(string sql)
{
    int result = 0;
    SqlConnection con = new SqlConnection(conStr);
    try
    {
        con.Open();
        SqlCommand cmd = new SqlCommand(sql, con);
        result = cmd.ExecuteNonQuery();
    }
    catch (Exception)                          //捕获错误信息
    {
        MessageBox.Show("系统繁忙！", "提示");
    }
    finally
    {
        con.Close();
    }
    return result;
}
```

注意：代码中用到的 GetStoreRoomByProt_Name(common.Prot_name)方法在前面已经讲过类似的代码，因此这里不做过多讲解。

19.5.14　实现销售出库记录查询

在前面实现商品销售时，当单击保存按钮后系统提示销售成功，与此同时系统也记录下了销售信息。在这里需要做的就是让存入的销售记录显示出来，也就是一条 SELECT 语句，仅此而已。具体实现代码如示例代码 19-30 所示。

示例代码 19-30

```
//窗体加载
private void Frm_Sell_Statistic_Out_Load(object sender, EventArgs e)
{
   dgv_Orders_Details.AutoGenerateColumns = false;
   dgv_Orders_Details.DataSource = GetOrders(2, dtp_Begin.Value, dtp_
   End.Value);
}
//显示订单明细
private void btt_Detailed_Click(object sender, EventArgs e)
{
   if (dgv_Orders_Details.Columns["cl_Order_sum_money"].Visible)
   {
```

```
        dgv_Orders_Details.Columns["cl_Order_sum_money"].Visible = false;
        dgv_Orders_Details.Columns["cl_Order_sum_total"].Visible = false;
        dgv_Orders_Details.Columns["cl_order_det_id"].Visible = true;
        dgv_Orders_Details.Columns["cl_Prot_name"].Visible = true;
        dgv_Orders_Details.Columns["cl_order_det_sum"].Visible = true;
        btt_Detailed.Text = "显示订单总表";
        gb_Statistic.Text = "销售退货订单明细表统计数据: ";
        dgv_Orders_Details.DataSource = null;
        dgv_Orders_Details.DataSource =
        GetOrderDetails(2, dtp_Begin.Value, dtp_End.Value);
    }
    else
    {
        dgv_Orders_Details.Columns["cl_Order_sum_money"].Visible = true;
        dgv_Orders_Details.Columns["cl_Order_sum_total"].Visible = true;
        dgv_Orders_Details.Columns["cl_order_det_id"].Visible = false;
        dgv_Orders_Details.Columns["cl_Prot_name"].Visible = false;
        dgv_Orders_Details.Columns["cl_order_det_sum"].Visible = false;
        btt_Detailed.Text = "显示订单表明细";
        gb_Statistic.Text = "销售退货订单总表统计数据: ";
        dgv_Orders_Details.DataSource = null;
        dgv_Orders_Details.DataSource = GetOrders(2, dtp_Begin.Value, dtp_
        End. Value);
    }
}
/// <summary>
/// 查询订单总表
/// </summary>
/// <param name="sql">sql 语句</param>
/// <param name="Order_type">该订单类型 0 进货 1 退货 2 销售 3 销售退货</param>
/// <param name="Begin">开始时间</param>
/// <param name="End">结束时间</param>
/// <returns></returns>
public List<Orders> GetOrders(int Order_type, DateTime Begin, DateTime End)
{
string sql =
            string.Format("select orders.*,userInfo.UserName from orders,
            userInfo where orders.UserId=userInfo.UserId and Ordertype={0}",
            Order_type);
    List<Orders> lo = new List<Orders>();
    SqlConnection con = new SqlConnection(DBHelper.conStr);
    con.Open();
    SqlCommand com = new SqlCommand(sql, con);
    SqlDataReader dr = com.ExecuteReader();
    while (dr.Read())
    {
        Orders o = new Orders();
        o.Order_id = Convert.ToInt32(dr["Orderid"]);
        o.Order_sum_money = Convert.ToSingle(dr["Ordersummoney"]);
        o.Order_sum_total = Convert.ToInt32(dr["Ordersumtotal"]);
        o.Order_time = Convert.ToDateTime(dr["Ordertime"]);
        o.Order_type = Convert.ToInt32(dr["Ordertype"]);
        o.UserId = Convert.ToInt32(dr["UserId"]);
        o.UserName = dr["UserName"].ToString();
        lo.Add(o);
    }
    dr.Close();
    con.Close();
```

```
    return lo;
}
/// <summary>
/// 查询订单总表或者明细表
/// </summary>
/// <param name="sql">sql 语句</param>
/// <param name="Order_type">该订单类型 0 进货 1 退货 2 销售 3 销售退货</param>
/// <param name="Begin">开始时间</param>
/// <param name="End">结束时间</param>
/// <returns></returns>
public List<OrderDetails> GetOrderDetails(int Order_type, DateTime Begin, DateTime End)
{
string sql =
            "select orders.*,orderDetails.*,userInfo.UserName,productIn-
            fo. Protname from orderDetails,orders,productInfo,userInfo
            where orderDetails.Orderid=orders.Orderid and orderedtails.
            Protid=productInfo.Protid and orders.UserId=userInfo.UserId
            and orders.Ordertype=@Order_type";
    List<OrderDetails> lo = new List<OrderDetails>();
    SqlConnection con = new SqlConnection(DBHelper.conStr);
    con.Open();
    SqlCommand com = new SqlCommand(sql, con);
    com.Parameters.Add("@Order_type", SqlDbType.Int).Value = Order_type;
    SqlDataReader dr = com.ExecuteReader();
    while (dr.Read())
    {
        OrderDetails o = new OrderDetails();
        o.Order_det_id = Convert.ToInt32(dr["orderdetid"]);
        //由于篇幅限制此处省略类似代码
         …
        o.UserName = dr["UserName"].ToString();
        lo.Add(o);
     }
     dr.Close();
     con.Close();
     return lo;
}
```

注意：这里主要讲解功能的实现思路，以上代码省略了异常处理。

19.5.15　实现销售退货记录查询

对于此功能的实现思路和方法与前面的销售出库记录查询基本相同，只是使用方法时传入的参数不同而已，因此不做过多讲解。这里简单地讲解一下使用方法时传入的参数不同的具体体现，具体代码如示例代码 19-31 所示。

示例代码 19-31

```
//显示订单信息
private void btt_Detailed_Click(object sender, EventArgs e)
{
   if (dgv_Orders_Details.Columns["cl_Order_sum_money"].Visible)
   {
```

```
        dgv Orders Details.Columns["cl Order sum money"].Visible = false;
        dgv_Orders_Details.Columns["cl_Order_sum_total"].Visible = false;
        dgv Orders Details.Columns["cl order det id"].Visible = true;
        dgv Orders Details.Columns["cl Prot name"].Visible = true;
        dgv Orders Details.Columns["cl order det sum"].Visible = true;
        btt_Detailed.Text = "显示订单总表";
        gb_Statistic.Text = "销售退货订单明细表统计数据：";
        dgv Orders Details.DataSource = null;
        //使用相同的方法，传入的参数不同实现的功能不同
        dgv Orders Details.DataSource =
        GetOrderDetails(3, dtp Begin.Value, dtp End.Value);
    }
    else
    {
        dgv Orders Details.Columns["cl Order sum money"].Visible = true;
        dgv Orders Details.Columns["cl Order sum total"].Visible = true;
        dgv_Orders_Details.Columns["cl_order_det_id"].Visible = false;
        dgv Orders Details.Columns["cl Prot name"].Visible = false;
        dgv Orders Details.Columns["cl order det sum"].Visible = false;
        btt_Detailed.Text = "显示订单表明细";
        gb_Statistic.Text = "销售退货订单总表统计数据：";
        dgv Orders Details.DataSource = null;
        //使用相同的方法，传入的参数不同实现的功能不同
        dgv Orders Details.DataSource =
        GetOrders(3, dtp Begin.Value, dtp End.Value);
    }
}
```

19.5.16　实现采购入库记录查询

对于此功能的实现思路和方法，与前面销售出库记录查询基本相同，只是使用方法时传入的参数不同，因此不做过多讲解。这里简单地讲解一下使用方法时传入的参数不同的具体体现，具体代码如示例代码 19-32 所示。

示例代码 19-32

```
private void btt Detailed Click(object sender, EventArgs e)
{
        If(dgv Orders Details.Columns["cl Order sum money"].Visible)
        {
             ...//此处和前面方法中相同的代码省略
            //使用相同的方法，传入的参数不同实现的功能不同
            dgv Orders Details.DataSource =
            GetOrderDetails(0, dtp Begin.Value, dtp End.Value);
      }
      else
      {
            ...//此处和上面方法中相同的代码省略
            //使用相同的方法，传入的参数不同实现的功能不同
            dgv Orders Details.DataSource =
            GetOrders(0, dtp Begin.Value, dtp End.Value);
      }
}
```

19.5.17　实现采购退货记录查询

对于此功能的实现思路和方法与上面的销售出库记录查询基本相同，只是使用方法时传入的参数不同而已，因此在此不做过多讲解。这里简单地讲解一下使用方法时传入的参数不同的具体体现，具体代码如示例代码 19-33 所示。

示例代码 19-33

```
//显示订单信息
private void btt_Detailed_Click(object sender, EventArgs e)
{
        If(dgv_Orders_Details.Columns["cl_Order_sum_money"].Visible)
        {
            ...//此处和上面方法中相同的代码省略
            //使用相同的方法，传入的参数不同实现的功能不同
            dgv_Orders_Details.DataSource =
            GetOrderDetails(1, dtp_Begin.Value, dtp_End.Value);
    }
    else
    {
            ...//此处和上面方法中相同的代码省略
            //使用相同的方法，传入的参数不同实现的功能不同
            dgv_Orders_Details.DataSource =
            GetOrders(1, dtp_Begin.Value, dtp_End.Value);
    }
}
```

至此，从客户提出需求，到分析需求，再到数据库设计和模块划分，系统用例分析直到编码，分析中提到的功能全部实现。本例只是从原理上讲解了一个软件项目的开发流程和具体过程，当然大型项目中会有所不同，但是本质的思想一定是一致的，只是表现出的方式有所区别。

19.6　项 目 小 结

本章讲解了一个进销存管理信息系统从无到有的完整过程。系统在功能上比较完善，包括了维护基本资料、采购入库、采购退货、销售出库、销售退货、库存查询和统计查询。

而维护基本资料中又包括了用户信息维护、商品信息维护、供应商信息维护、品牌信息维护，统计查询中包括了销售出库记录查询、销售退货记录查询（包括退货原因）、采购入库记录查询、采购退货记录查询（包括退货原因）等。尽管如此但系统还是有一些不完善的地方，比如打印采购单和销售单等。读者可以在加深理解后，对系统进行进一步的完善。

本章详细介绍了进销存系统的开发过程，并给出了部分源代码，主要内容如下：

- ❑ 进销存系统的需求分析。
- ❑ 进销存系统的系统分析。
- ❑ 进销存系统的功能模块分析。

- ❑ 进销存系统的业务流程分析。
- ❑ 进销存系统的界面操作流程分析。
- ❑ 进销存系统的数据库分析和设计。
- ❑ 进销存系统的数据库连接、功能模块设计及代码实现。

本系统在开发过程中的亮点：

- ❑ 根据各个模块的功能需求，设计编写了完善的实现方法，并且运用了面向对象的思想，使得功能的实现思路非常清新，系统易于维护和扩展。
- ❑ 封装了数据库连接字符串。
- ❑ 操作流程清晰，通过使用的流程来贯穿整个的设计过程。

第 20 章　LINQ to SQL 实例——宾馆管理信息系统

随着我国企业信息化建设进程的加快和计算机的普及，使用计算机软件进行企业信息化管理，计算机代替手工操作已经成为现实和必然的发展趋势。当然作为酒店宾馆类服务性行业更是如此，客户量不断增大，服务项目增多、组织也变得庞大起来，所以要想提高劳动效率、降低运营成本、提高服务质量和管理水平，必然需要借助计算机进行管理。本项目正是在这种环境下应运而生。

20.1　项 目 概 述

受某宾馆委托，本软件公司按照其要求开发一套宾馆常用业务管理系统，以便给客户提供快速、便捷、准确和可靠的服务，并且要简单易用。

20.1.1　功能概述

对于前台来说，系统主要是为客人提供准确的信息，为客人办理入住手续，将客户信息录入电脑，并可以进行为客户调换房间、结账等操作。另外，根据宾馆的发展需要，还能进行调整房间类型价格、增加设施等操作，以便满足不同客户的需要。总体可以分为下面几种功能：

- ❑ 客房类型及状态维护。
- ❑ 客房状态查询。
- ❑ 客户等级维护。
- ❑ 楼层维护。
- ❑ 入住和结账。
- ❑ 入住数据查询。

20.1.2　用户环境描述

采取分布式架构，整个项目需要一个主服务器，每个楼层都至少需要有一台终端，24 小时不间断服务。

20.1.3　可行性分析

从投资角度分析，本项目能够提升企业的形象和市场竞争力，进行全方位计算机管理，

同时旨在提升客户满意度和服务质量，辅助高层进行市场应对决策。

从技术可行性角度分析，目前.NET 平台已经非常成熟，并且 Ms SQL Server 2008 已经成为各个大中型企业优先采用的数据库服务器，完全可以满足本宾馆客户需要。

从社会可行性角度看，采用软件管理业务已经成为一种绝对趋势，是先进管理思想的一种体现，并且操作员为了减少手工记账的错误也乐于接受计算机管理的方便快捷。

20.2　项目需求分析

需求是任何一个项目成败的根本，需求分析的目的是规范化本项目的开发，旨在提高软件开发过程中的能见度，便于客户和程序员之间交流、协作并作为工作成果的原始依据，同时也表明了本项目的共性，以期能够得到更大范围的应用。通过和客户的反复沟通及确认，最终的总业务流程如图 20-1 所示。



图 20-1　总流程图

20.2.1　系统功能性需求分析

根据客户的要求，从功能的角度需要分析确定，以便于模块划分。功能性需求分析的过程其实就是将自然语言转换为模块或者功能的过程，明确区分每个业务的边界和目的及过程，是一个再次细化的过程，分析结果如表 20-1。

表 20-1　宾馆管理系统功能说明

功 能 类 别	功能名称、标识符	描　　述
系统资料维护	房间类型维护	操作员对房间类型的增删改查等操作
	楼层维护	操作员对本宾馆楼层的增删改查等操作
	房间维护	根据上面的楼层和房间类型，增加或修改删除房间信息
	客户分类维护	操作员根据宾馆规定对客户进行分级操作和相应的折扣操作
	房间状态维护	操作员对房间状态进行修改
入住登记	客户入住登记操作	登记客户信息，分配相应的房间号，修改房间状态第一次入住需要登记，以后只需要查询
房间置换	房间置换操作	根据客户需要提供房间置换功能，置换需要结账
结账操作	客户结账操作	客户离开宾馆之前的结账操作，需要修改房间状态
查询操作	查询本月收益	查询本月宾馆总收入情况和入住率等
	查询入住明细	根据客户提供的身份证信息查询客户入住明细

20.2.2　系统总用例分析

功能分析清楚之后，重要的就是把心态功能划分成为相对独立的模块或者子系统，便于集中讨论和确定需求，并且需要确定出操作用户。下面就根据功能用例图的方式直观地表现出该系统的业务流程，如图 20-2 所示。

20.2.3　系统用例分析

【用例 1：房间类型维护】

- ❑ 描述：系统正式上线时执行增加操作，一般类型名称为标准间、普通间、总统套间和豪华间等。操作员需要根据宾馆实际情况，对这些信息进行维护操作（注意删除操作，已经使用的房间类型不能执行删除操作）。
- ❑ 参与者：操作员。
- ❑ 用例图：图 20-3。

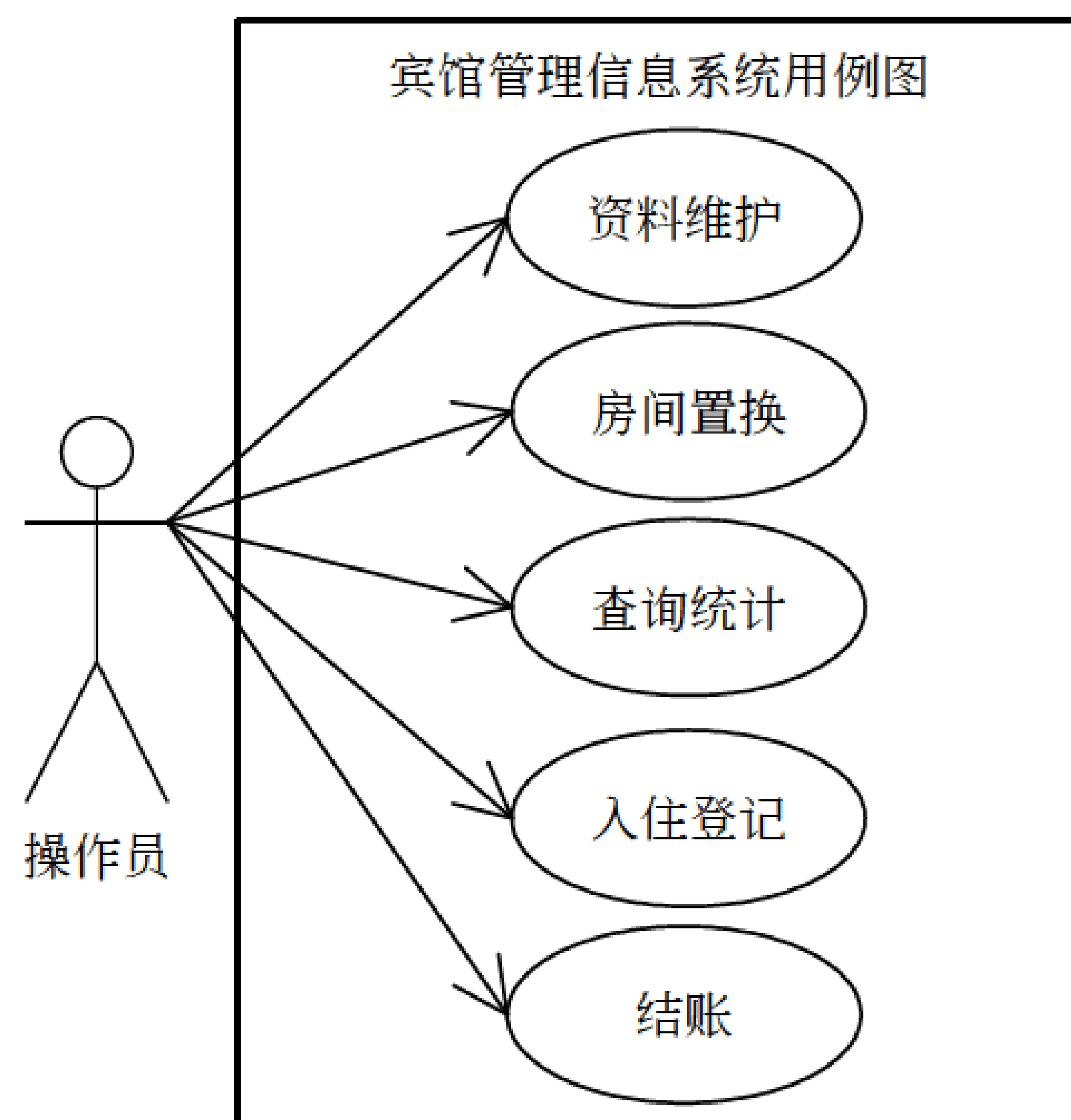


图 20-2　系统总用例

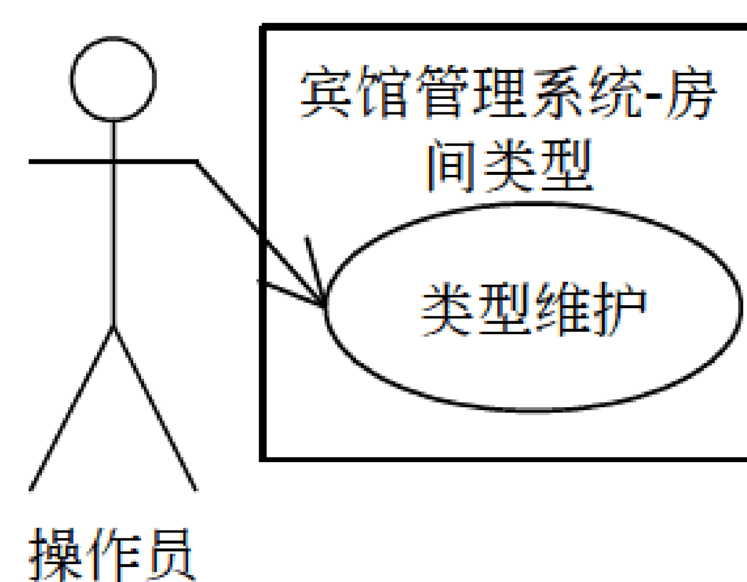


图 20-3　房间类型用例图

【用例 2：楼层维护】

- ❑ 描述：系统正式上线时维护。用于标识楼层信息，一般设置为一楼、二楼等，特殊的还可以设置成偏楼一楼等信息。操作员根据宾馆实际情况设置（注意删除操作，已经使用的楼层信息不能删除）。
- ❑ 参与者：操作员。
- ❑ 用例图：图 20-4。

【用例 3：房间维护】

- ❑ 描述：包含宾馆的所有房间信息，主要描述房间号，类型、状态和床位设置及房间设施备注信息。操作员可以根据宾馆实际情况对房间信息进行增加、修改、查询和删除操作。
- ❑ 参与者：操作员。
- ❑ 用例图：图 20-5。

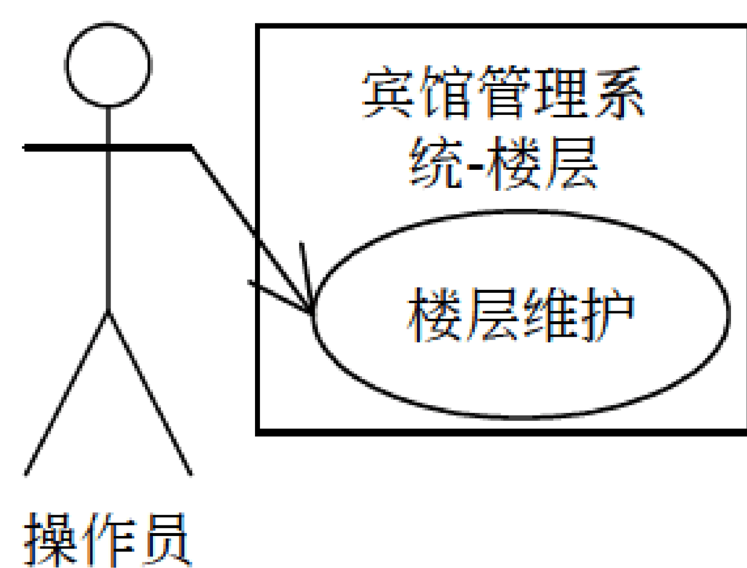


图 20-4　楼层维护

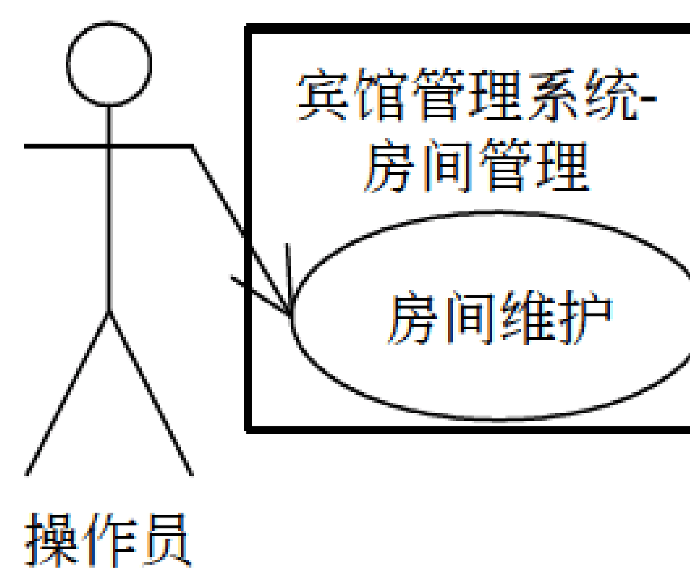


图 20-5　房间维护

【用例 4：客户分类维护】

- 描述：针对常住老客户或者重要客户需要设置打折优惠，本操作目的是为不同级别的客户设置优惠信息。一般可以设置为金卡、银卡或 VIP 等类型信息，同时设置不同等级对应的折扣情况。操作员可以根据实际情况调整客户等级信息。
- 参与者：操作员。
- 用例图：图 20-6。

【用例 5：客户分类维护】

- 描述：宾馆的房间需要明确显示多种状态，方便前台服务人员决策。客户入住就需要把该房间状态设置为入住，说明此时房间不能再安排别人，否则可以设置为空闲。当然还有维修、打扫、自用等多种状态。需要提前维护好，形成固有名词，后续直接使用。凡是非“空闲”状态，均不可再安排客人入住，需要每个楼层的服务员在实施“打扫”后把状态更新过来，方可使用。
- 参与者：操作员。
- 用例图：图 20-7。

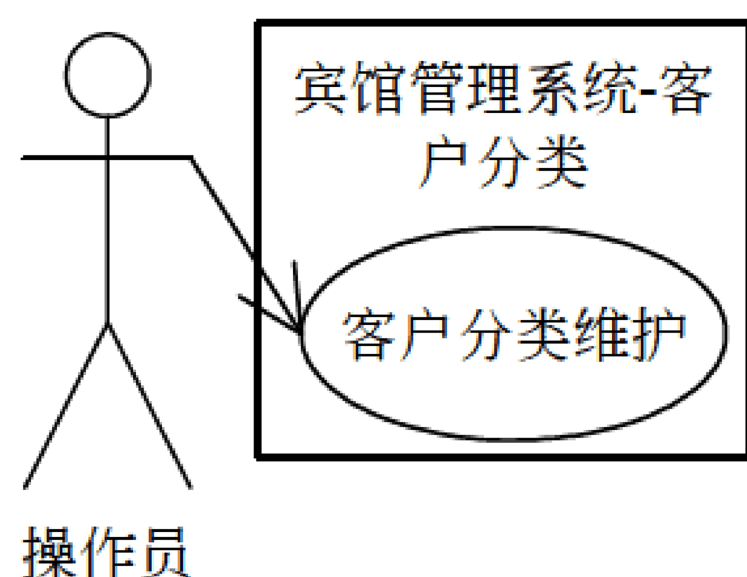


图 20-6　客户分类

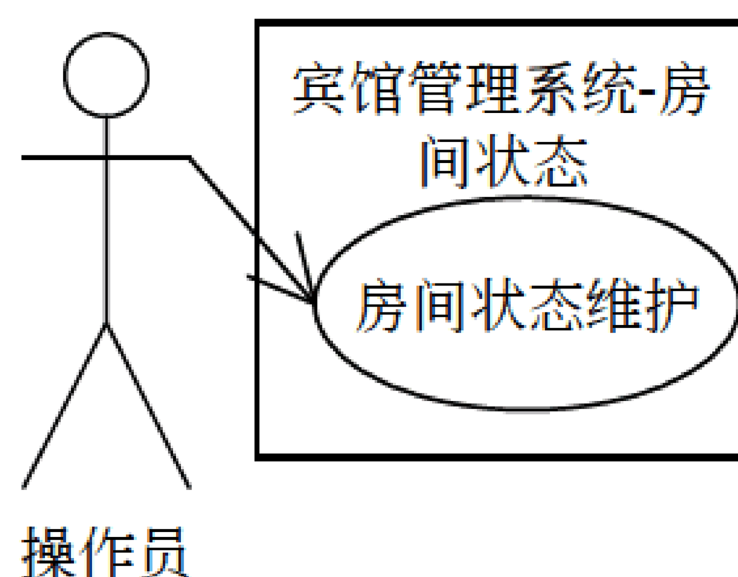


图 20-7　房间状态

【用例 6：客户入住登记】

- 描述：用于客人开房的登记，需要由操作员录入客户信息，在录入时需要区分企业客户还是个人客户。如果是个人客户需要检测身份证是否已经存在，如果存在则显示客户信息直接登记，否则重新录入。如果是企业客户，需要查询企业名称是否存在，如果存在，则显示该企业客户信息直接登记；否则重新录入。另外房间状态为非“空闲”的不可安排入住（注意：本次项目不涉及客户预交款及财务业务，所以入住暂时不考虑预交款事宜）。
- 参与者：操作员。
- 用例图：图 20-8。

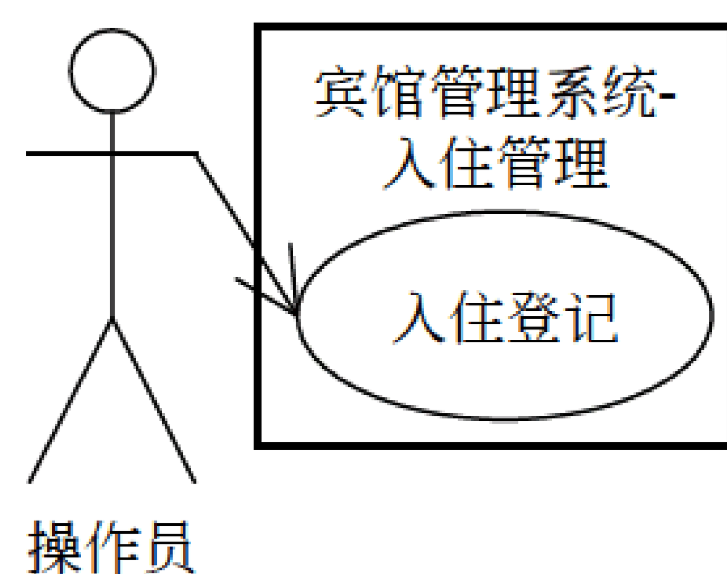


图 20-8　入住登记

【用例 7：房间置换】

- 描述：客户由于一些原因需要置换房间，执行的流程是先给原来的房间结账，再登记新的房间。置换的前提是新房间必须为“空闲”状态，置换后的老房间需要更新为“打扫”状态，以便供下个客户入住。置换后的房间收费与新房间类型同步计算。
- 参与者：操作员。
- 用例图：图 20-9。

【用例 8：客户结账】

- 描述：客户需要离开宾馆时必须先结账，根据入住时间和房间类型计算消费金额，结账后必须把房间状态更新为“打扫”。
- 参与者：操作员。
- 用例图：图 20-10。

【用例 9：查询业务】

- 描述：宾馆决策层需要及时了解宾馆统计信息，如本月收益和入住率等，有时还要根据需要查询客户入住明细。
- 参与者：操作员。
- 用例图：图 20-11。

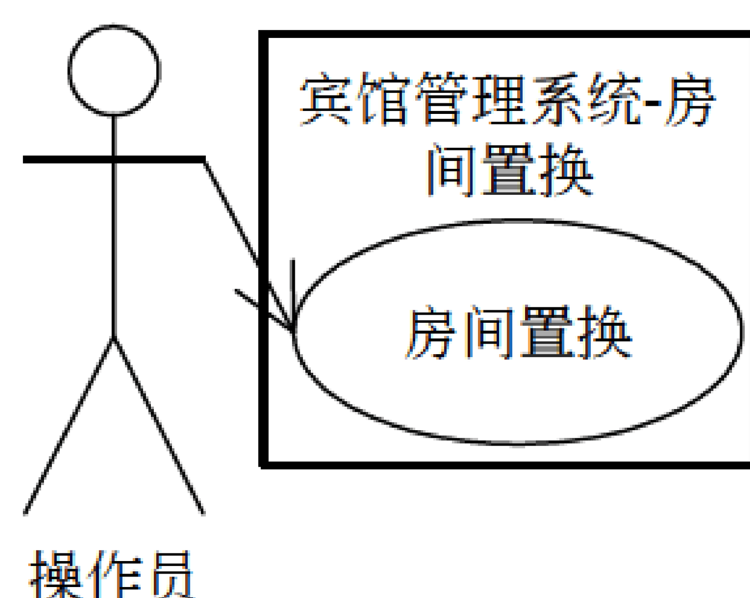


图 20-9　房间置换

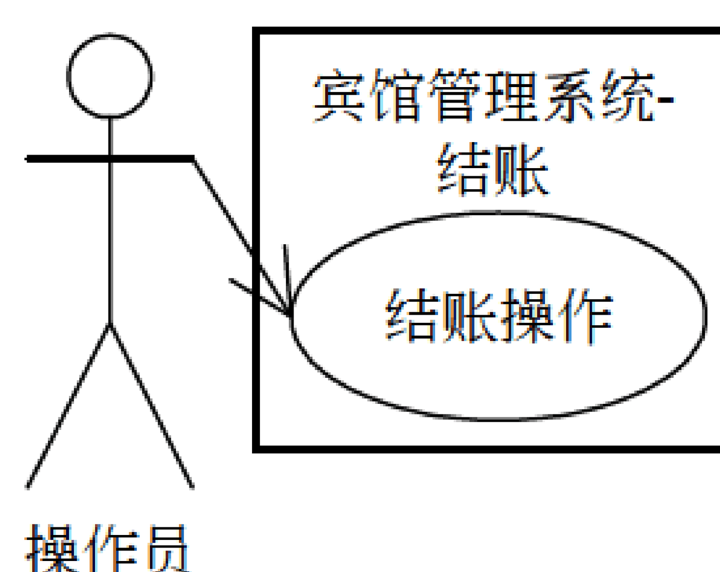


图 20-10　客户结账

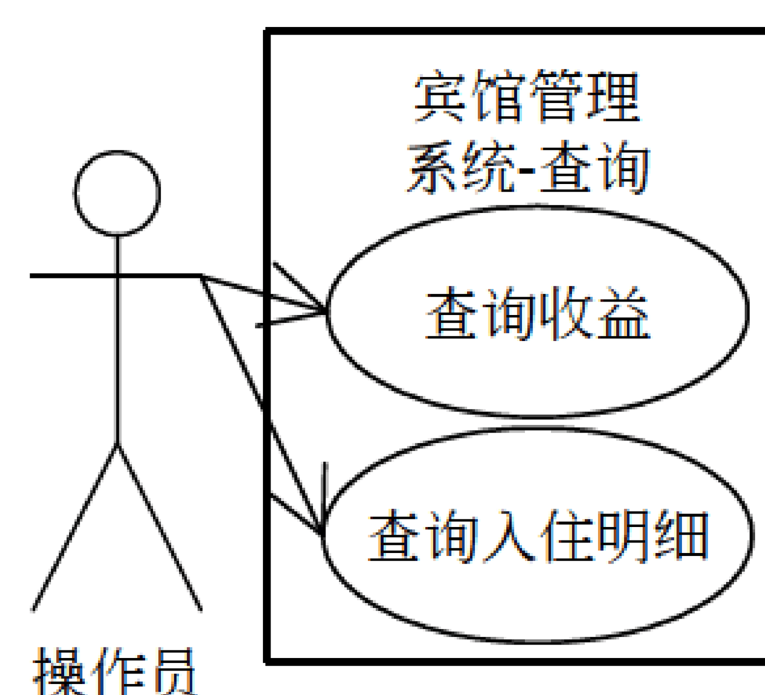


图 20-11　查询操作

20.3　系统总体架构设计

在软件组织中，系统架构无疑是最重要的环节之一。一般认为，一个设计如果必须要求高手云集才能生产出符合质量要求的产品，并不一定是好的架构。架构设计有一个很重要的目标就是能够使用总体上能力一般的队伍，通过组织和设计的力量完成复杂并且难度较大的任务。另一方面，由于客户方的需求变更，必然会导致设计的调整而造成开发成本倍增，同样由于系统维护难度较大导致投入再次增加，从投资回报的角度来说，是任何人都不愿意看到的。所以必须研究系统架构，以便能够设计出适应系统变更、维护与升级，同时要尽可能地节约成本的架构来。

在软件体系架构设计中，分层式结构最常见，流行三层开发架构（3-tier application）的居多，也是最重要的一种结构。微软推荐的分层式结构一般分为三层，从下至上分别为：数据访问层、业务逻辑层（又或称为领域层）、表示层。区分层次的目的是为了“高内聚，低耦合”的思想。架构结构图如图 20-12 所示。

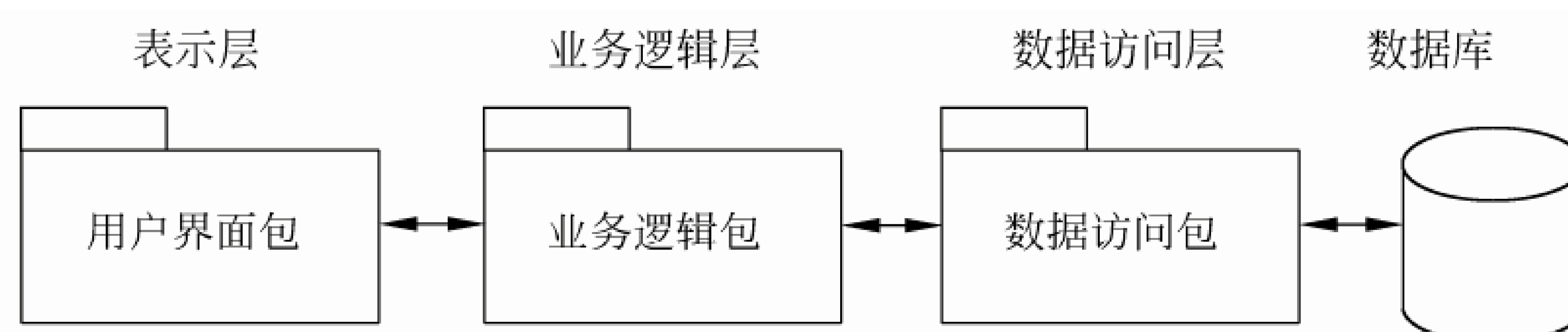


图 20-12　分层架构图示

20.3.1　三层结构原理

三个层次中，系统主要功能和业务逻辑都在业务逻辑层进行处理。所谓三层体系结构，是在客户端与数据库之间加入了一个“中间层”，也叫组件层。这里所说的三层体系，不是指物理上的三层，不是简单地放置三台机器就是三层体系结构，也不仅仅有 B/S 应用才是三层体系结构，三层是指逻辑上的三层，即使这三个层放置到一台机器上。三层体系的应用程序将业务规则、数据访问、合法性校验等工作放到了中间层进行处理。通常情况下，客户端不直接与数据库进行交互，而是通过 COM/DCOM 通信与中间层建立连接，再经由中间层与数据库进行交互。最后在面向对象领域，三层架构有了进一步的演化。由于面向对象编程把事物当成对象对待，各个系统中也不例外，所有描述的均是对象，系统管理的也是对象，而在编程中对象是需要先实例化出来才能使用。为了描述系统管理的每类对象，需要针对每一个实体建立一个描述类，这个描述类放在上三个层中都不合适，所以需要重新构建一个层，存储实体描述类，称之为“实体层”。各层功能分析如下所示。

- 表示层：位于最外层（最上层），离用户最近。用于显示数据和接收用户输入的数据，为用户提供一种交互式操作的界面。
- 业务逻辑层：业务逻辑层（Business Logic Layer）无疑是系统架构中体现核心价值的部分。它的关注点主要集中在业务规则的制定、业务流程的实现等与业务需求有关的系统设计。也即是说它是与系统所应对的领域（Domain）逻辑有关，很多时候，也将业务逻辑层称为领域层。例如 Martin Fowler 在 *Patterns of Enterprise Application Architecture* 一书中，将整个架构分为三个主要的层：表示层、领域层和数据源层。作为领域驱动设计的先驱 Eric Evans，对业务逻辑层作了更细致的划分，细分为应用层与领域层，通过分层进一步将领域逻辑与领域逻辑的解决方案分离。业务逻辑层在体系架构中的位置很关键，它处于数据访问层与表示层中间，起到了数据交换中承上启下的作用。由于层是一种弱耦合结构，层与层之间的依赖是向下的，底层对于上层而言是“无知”的，改变上层的设计对于其调用的底层而言没有任何影响。如果在分层设计时，遵循了面向接口设计的思想，那么这种向下的依赖也应该是一种弱依赖关系。因而在不改变接口定义的前提下，理想的分层式架构，应该是一个支持可抽取、可替换的“抽屉”式架构。正因为如此，业务逻辑层的设计对于一个支持可扩展的架构尤为关键，因为它扮演了两个不同的角色。对于数据访问层而言，它是调用者；对于表示层而言，它却是被调用者。依赖与被依赖的关系都纠结在业务逻辑层上，如何实现依赖关系的解耦，则是除了实现业务逻辑之外留给设计师的任务。
- 数据层：数据访问层：有时也称为是持久层，其功能主要是负责数据库的访问，可以访问数据库系统、二进制文件、文本文档或是 XML 文档。简单的说法就是实现对数据表的 Select、Insert、Update、Delete 的操作。如果要加入 ORM 的元素，那么就会包括对象和数据表之间的 mapping，以及对象实体的持久化。
- 实体层：简单地说，就是用来描述管理对象的，由字段和封装后的属性组成，然后用实体类描述的对象进行数据传输和管理。也可以说实体类就是一个数据的载体，使用实体类可以消除各个关系数据库和对象之间的差异，其具备面向对象的

基本特征，是一个完全受控的对象。

20.3.2　系统三层搭建

按照宾馆管理信息系统的要求，本项目需要一个客户操作界面。由于在局域网内使用，所以首选.NET 平台下大家都熟悉的 WinForm 界面，负责和操作员交互，接收输入和显示输出。应该划分为表示层，表示层的搭建方法如下。

（1）创建表示层：首先打开 Visual Studio 2010，新建项目。在弹出的“新建项目”对话框项目类型中选择 Visual C#|Windows 选项，模板选择“Windows 窗体应用程序”，填写项目名称“HotelMis”，并选中“为解决方案创建目录”复选框，如图 20-13 所示。

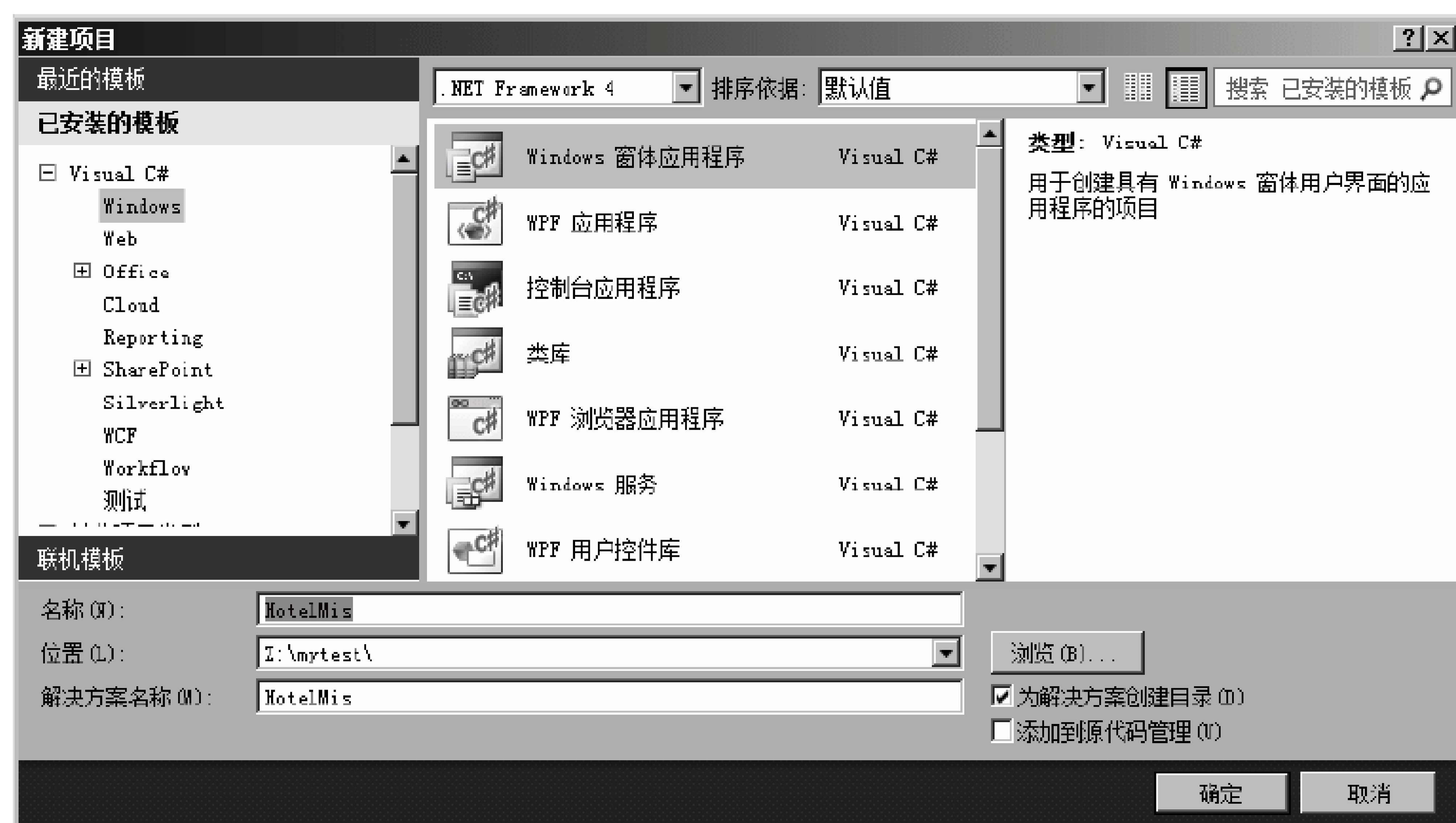


图 20-13　创建表示层

（2）创建业务逻辑层：在创建好的解决方案名称上右击，在弹出的快捷菜单中选择“添加”|“新建项目”命令，打开“添加新项目”对话框。在模板中选择“类库”选项，填写名称“HotelMis.BLL”，如图 20-14 所示。



图 20-14　创建业务逻辑层

（3）创建数据访问层：创建数据访问层和创建业务逻辑层的方法相同，只是在“名称”

文本框中输入“HotelMis.DAL”，如图 20-15 所示。



图 20-15　创建数据访问层

（4）创建实体层：创建实体层和创建业务逻辑层的方法相同，只是在“名称”文本框中输入“HotelMis.Models”，如图 20-16 所示。



图 20-16　创建实体层

创建 LINQ 实体层一般采用工具生成，可以用 Visual Studio 自带的映射工具，操作步骤：选择“工具”|“连接到数据库”命令，会弹出如图 20-17 的对话框。

填写相应的数据库服务器名（本机服务器名可以用实心圆点表示），然后选择要使用的数据库确定。右击实体层，在弹出的快捷菜单中选择“添加”|“新建项”命令，在打开的对话框中选择第 2 个“LINQ tO SQL 类”，在“名称”文本框中输入名称，单击“添加”按钮，如图 20-18 所示。系统会自动打开一个设计视图。

然后在左边浮动窗体的“服务资源管理器”中，找到刚才连接好的数据库，展开表结点。全部选中，拖放到主操作区的设计视图中，结果如图 20-19 所示。



图 20-17　选择数据库



图 20-18　添加 LINQ to SQL 类

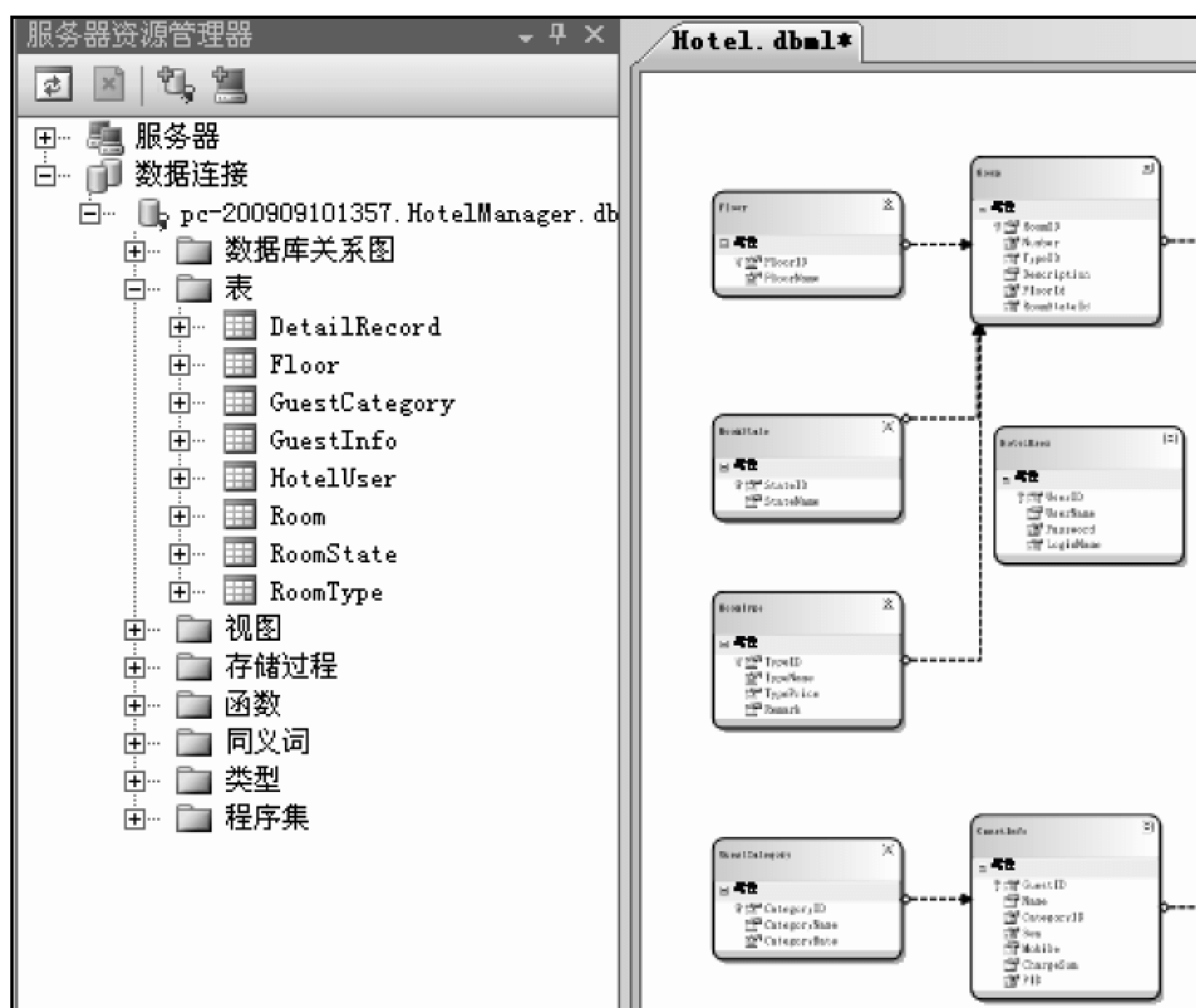


图 20-19　生成关系图

添加引用：三层的框架已经搭建成功，但是每层之间是互相独立的。.NET 应用程序中需要对层之间添加引用，创建依赖关系。

（1）实现表示层对业务逻辑层的引用：打开“解决方案资源管理器”面板，右击表示层的“引用”图标，在弹出的快捷菜单中选择“添加引用”选项，如图 20-20 所示。打开“添加引用”对话框，如图 20-21 所示。选择“项目”选项卡，在其选中名称“BLL”，单击“确定”按钮即可。此时在表示层的引用目录中可以看到刚才添加的 BLL，如图 20-22 所示。

（2）实现业务逻辑层对数据访问层的引用和上述表示层引用业务逻辑层一样。最后上

述三层都要引用试题层，需要再次添加 3 遍，最终引用的结果如图 20-23 所示。



图 20-20　添加引用页

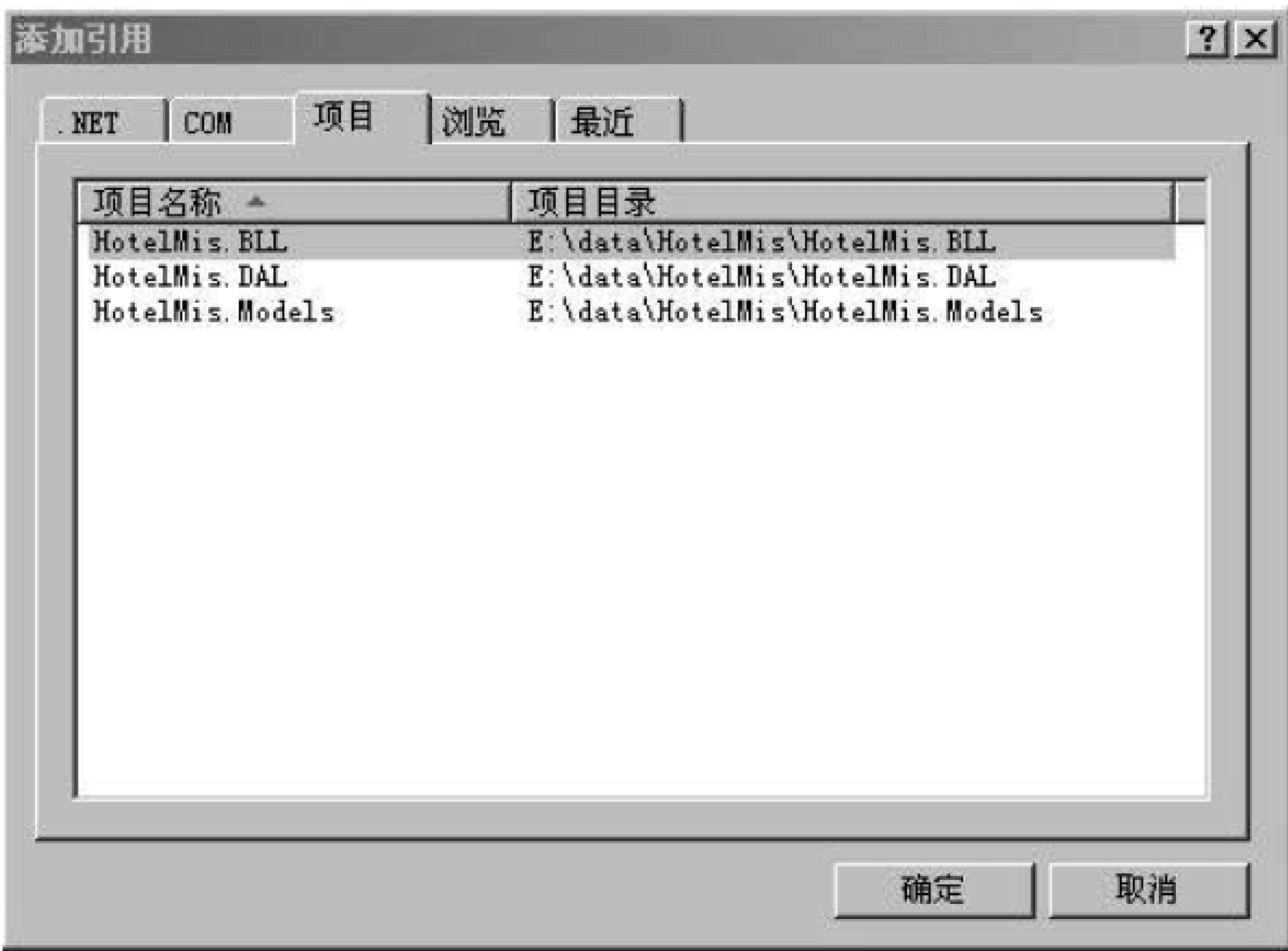


图 20-21　选择引用项目

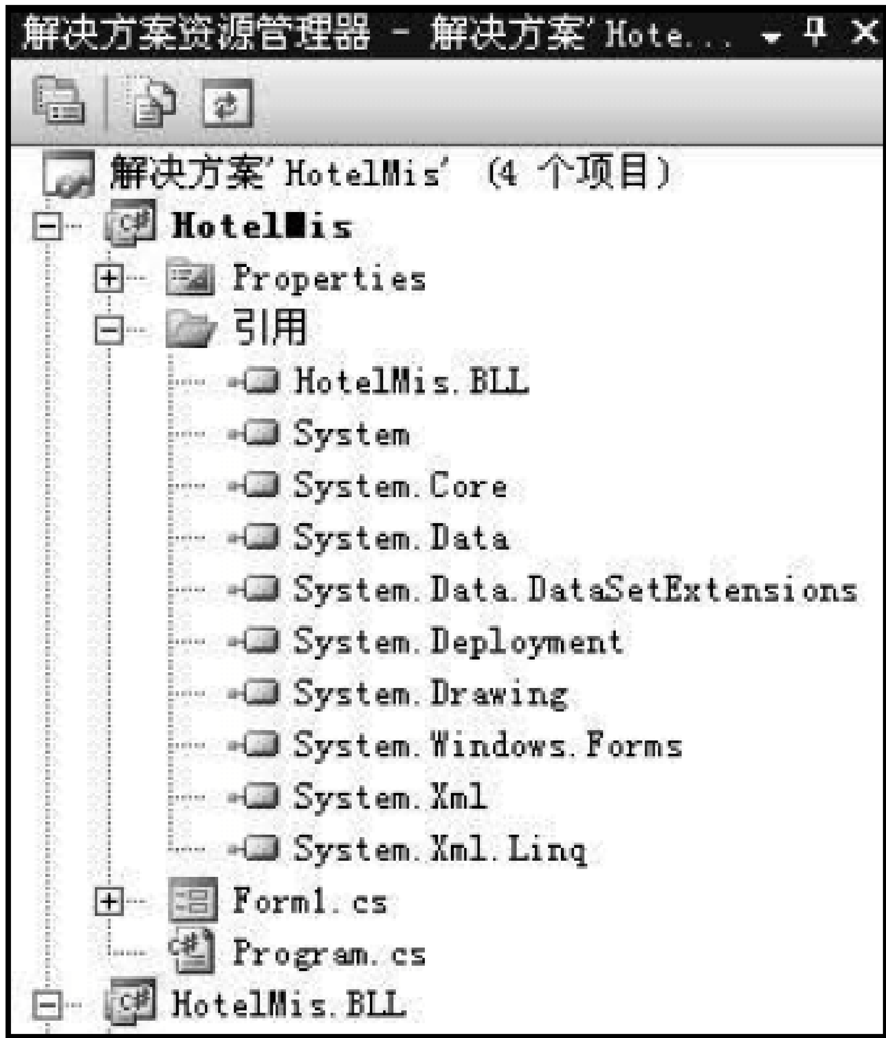


图 20-22　添加引用后

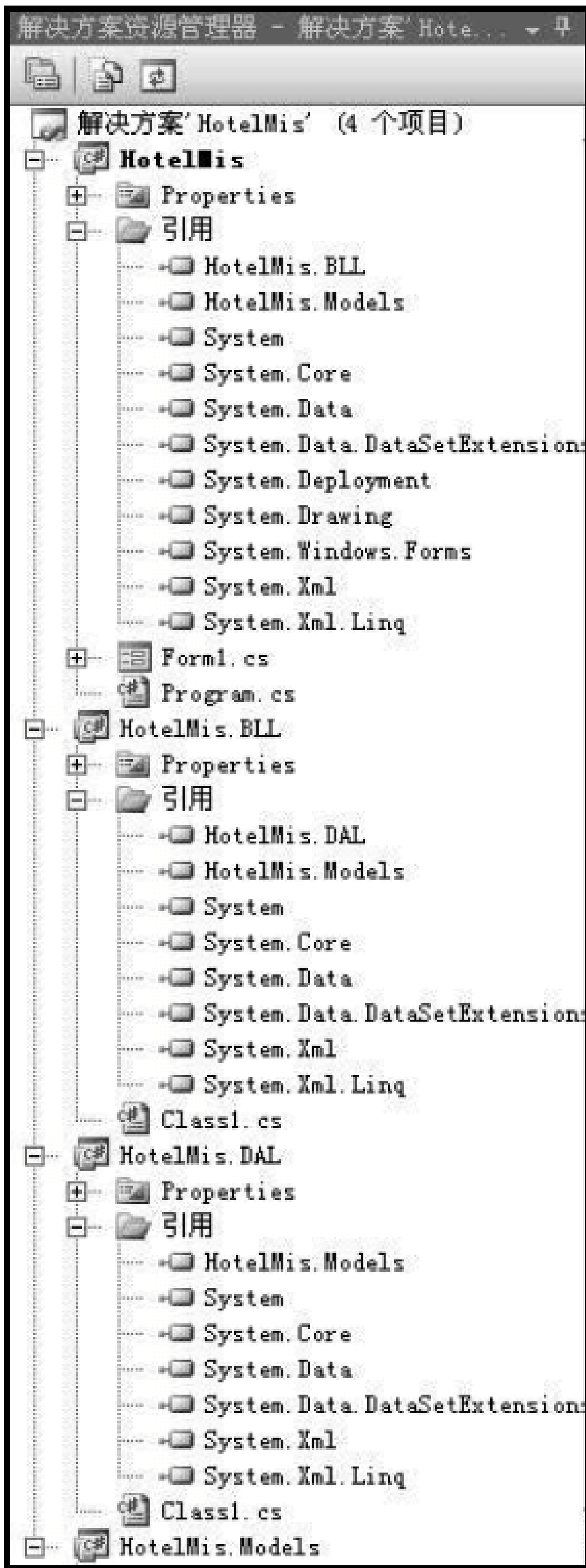


图 20-23　完整引用图

注意： 上面在搭建三层架构时，推荐的命名方式是“项目名+.+DAL”，原因是为了后续代码编写中引用方便，可以直接用项目名点出所有层。当然读者也可以不写这个“.”只需要在项目上右击，在弹出的快捷菜单中选择“属性”，在属性面板上修改命名空间，在 DAL 前加个点，效果是一样的。

20.4　系统数据库设计

数据库应该是一个企业最重要的部分，建立企业自己的额度数据库就成了非常关键的一个环节。对于程序来说，安全、完整地保存客户信息就成了义不容辞的责任，因此数据库设计就变为整个软件开发过程中最重要的环节之一。一旦数据设计出现问题，就相当于软件的根基出了问题，流程就会受到影响。所以，数据库的设计工作需要非常严谨态度和科学的做法。本节将重点介绍数据库设计的相关步骤和方法。

20.4.1　收集客户信息

在 20.2 项目需求分析阶段已经收集到软件相关需求，这些可以作为数据库设计的信息来源。注意要把客户要求的功能点按照完整的一句话表述清楚，描述中的名词要准确，动词对应对象也要明确。比如“操作员维护房间状态”就可以把“谁”做“什么”，“怎么做”表达清楚。整理后的功能信息如下：

- 操作员负责客房类型及状态维护。
- 操作员可以对客房状态查询。
- 操作员进行客户等级维护。
- 操作员对楼层进行维护。
- 操作员安排客户入住和结账。
- 操作员对客户入住数据进行查询。

20.4.2　标识对象

经过上面的分析可以确定存储对象，但是距离数据库存储还比较远。接下来的任务就是找出要准备存储谁。一个比较常用的做法是找出需求中的关键名词，如上述分析中的操作员、客户、房间等就是潜在的存储对象，这些名词表述要一致、准确。按照这个思路可以标识出这些对象：操作员，客房，客户，而所谓的“类型”等信息是附加在名词上的修饰部分，本阶段不予分析。

20.4.3　标识对象的属性

存储对象标识出来之后，要把这些“名词”存进数据库，显然不可能，第三代关系型数据库存储的是对象的描述部分，也就是说把对象的关键描述点抽取出来，存入数据库来作为对这个对象的描述。而这些关键描述点就是通常说的“属性”。属性的抽取方法主要是从客户需求而来。比如存储学生信息，关键点在于学生的年龄、性别、家庭住址等，和该学生的肤色、长相关系不是很密切。按照这个要求抽取的结果如下。

- 操作员：用户名和密码。
- 客户：名称、性别、身份证、级别、电话和消费。

❑ 房间：类型、房间号、状态、楼层和描述。

20.4.4　标识对象之间的关系

任何一个系统中存储和管理的对象都不是相互独立的，都有一定的联系和关联，分析清楚关系，有助于分析业务和数据流向。那么操作员和客户的关系就很明确了——录入和查询（一对多），操作员和房间的关系——维护（一对多），客户和房间的关系——入住（一对一）。

20.4.5　系统 E-R 图

通过上述 4 步，基本可以把存储对象分析出来，最后要对分析的结果进行评估，稍微复杂一点的系统用文字描述就显得不够清晰了，这时就需要使用通用的实体关系图，也就是 E-R 图来表达了。结合上述分析，构建出的 E-R 图如图 20-24 所示

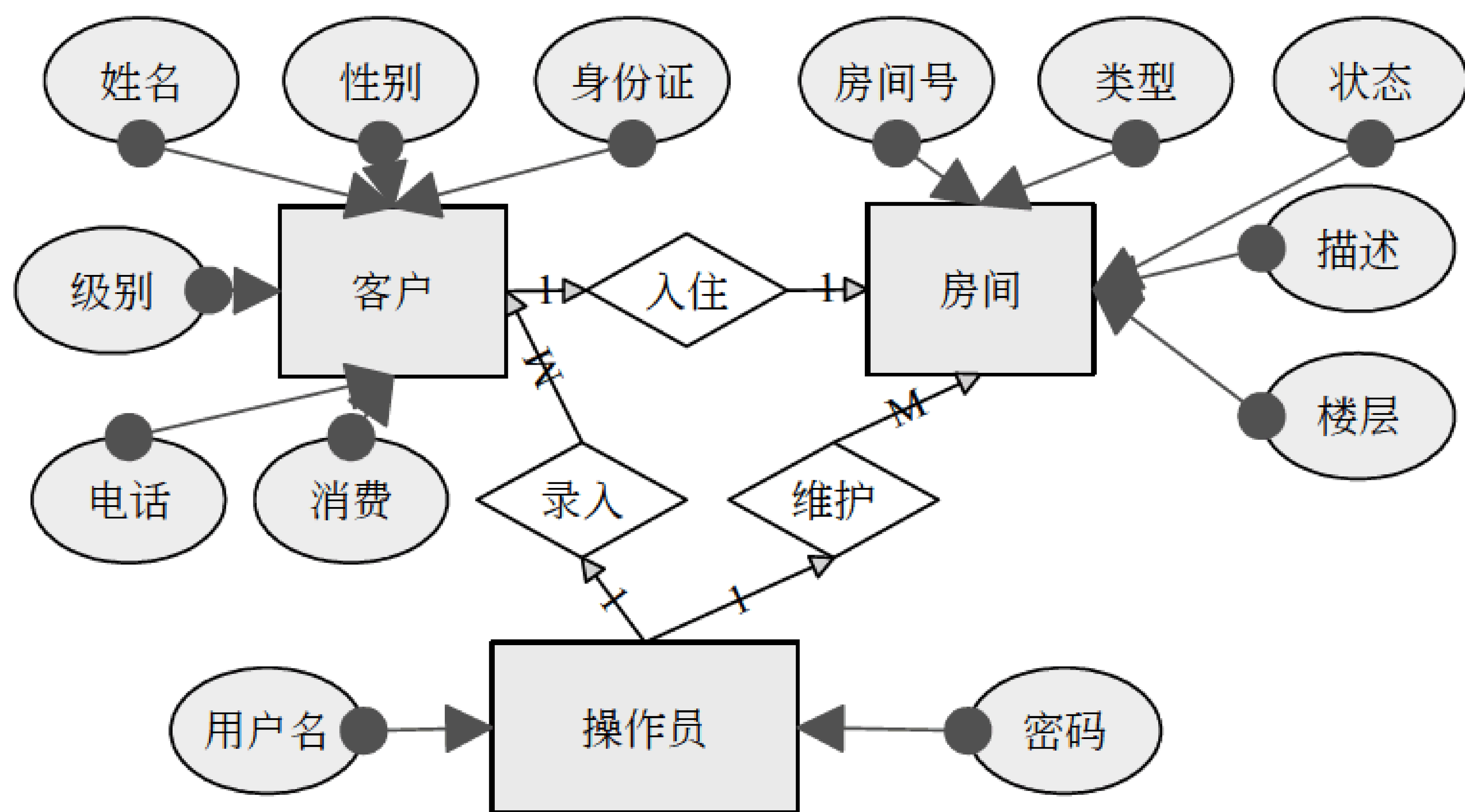


图 20-24　系统实体关系图

注意：绘制 E-R 图的方式有很多种，可以采用的软件有 PowerDesigner、E-rWin、Visio 等。

20.4.6　数据库逻辑设计

接下来的工作就是提取数据库模型了，也就是逻辑实现阶段，这个阶段需要注意的是如何使用三大范式约束逻辑设计。一般遵循 3 个原则即可：

❑ 每个字段都必须是不可再分的。
❑ 非主键字段必须完全依赖该主键。
❑ 有关键关系的表使用主外键联系起来。

最终形成翻译后的数据库关系图，如图 20-25 所示。

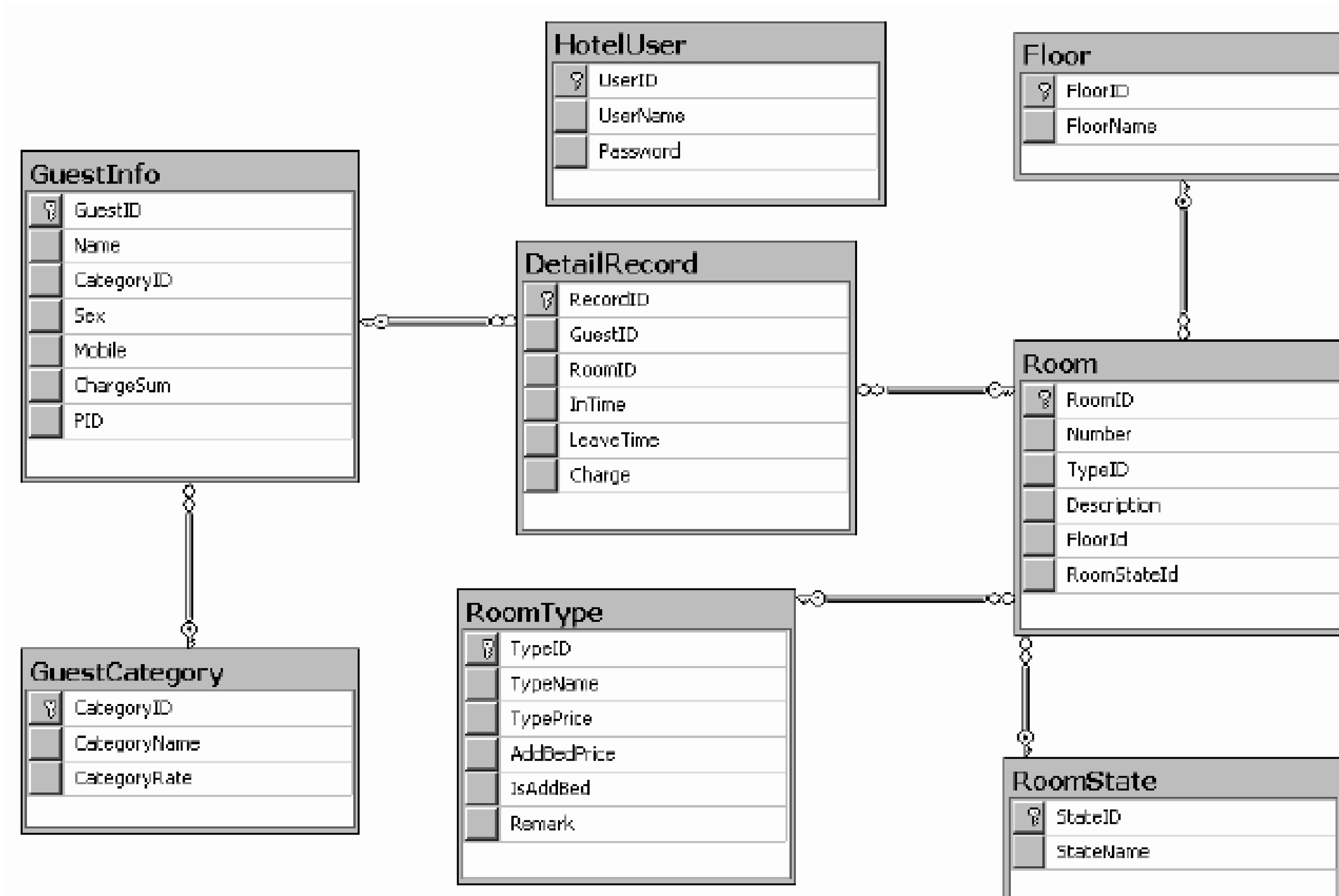


图 20-25　数据库关系图

20.4.7　表设计

由于本案例采用的是 SQL Server 2008 数据库，就需要按照 SQL Server 2008 的数据类型和规范把 20.4.6 节的逻辑设计反映成为物理设计就是创建物理表的过程，需要确定每列的数据类型、长度和约束。另外，每个字段是否可以为空，长度有没有限制等均需要和具体需求对照确定。约定数据库名字为 HotelManager，具体表设计如表 20-2 所示。

表 20-2　宾馆信息管理系统数据库HotelManager的表定义

属性 表名	字　段	类　型	描　述
DetailRecord（入住记录表）	RecordID	Int	记录唯一编号，自增，主键
	GuestID	Int	客户编号，外键，必填
	RoomID	Int	房间 ID，非空
	InTime	DateTime	入住时间
	LeaveTime	DateTime	离开时间
	Charge	Money	消费金额
Floor（楼层表）	FloorID	Int	楼层唯一编号，自增，主键
	FloorName	NVarChar(20)	楼层名称，非空
GuestCategory（客户类别表）	CategoryID	Int	类别编号，自增，主键
	CategoryName	NVarChar(10)	类别名称，非空
	CategoryRate	Float	折扣，非空
GuestInfo（客户信息表）	GuestID	Int	客户编号，自增，主键
	Name	NVarChar(10)	客户名称，非空
	CategoryID	Int	类别编号，非空，外键
	Sex	NVarChar(5)	性别
	Mobile	NVarChar(30)	电话
	ChargeSum	Money	消费总金额
	PID	NVarChar(20)	身份证号

续表

属性 / 表名	字　段	类　型	描　述
HotelUser（操作员表）	UserID	Int	操作员编号，自增，主键
	UserName	NVarChar(20)	操作员姓名，非空
	Password	NVarChar(20)	密码，非空
Room（房间信息表）	RoomID	Int	房间编号，自增，主键
	Number	NVarChar(20)	房间号，非空
	TypeID	Int	类型编号，外键
	Description	NVarChar(255)	描述信息
	FloorId	Int	楼层编号
	RoomStateId	Int	状态编号
RoomType（房间类型表）	StateID	Int	状态编号，自增，主键
	StateName	VarChar(10)	状态名称
RoomType（类型信息表）	TypeID	Int	类型编号，自增，主键
	TypeName	NVarChar(10)	类型名称，非空
	TypePrice	Money	类型价格，非空
	AddBedPrice	Money	加床价格
	IsAddBed	bit	是否允许加床
	Remark	NVarChar(255)	备注信息

注意：这里的数据类型和长度是根据实际调查和与客户沟通确定的，包括专有名词也来源于现实中，再一次说明了需求的重要性。

20.5　宾馆管理系统界面设计

一个好的应用程序不但要有完善的功能，还要有符合用户使用习惯的界面。用户界面是应用程序的一个重要组成部分。用户界面不仅影响到软件外观，而且对应用程序的易用性和可操作性都起了决定性作用。界面的设计应该从用户角度出发，以方便用户使用作为根本目标，因为整个软件使用过程中，界面和用户交互的时间是最长的。

20.5.1　界面设计标准

由于软件界面在整个软件生命周期内的重要作用，所以必须根据社会工程学、国家标准等相关规范，确定软件界面时要遵循以下原则：

- 布局合理，界面清洁。
- 配色合理，图像和显示效果要统一。
- 整个软件界面风格应该保持一致。
- 减少用户的操作负担，界面尽可能少地使用鼠标。

❑ 所有界面文字清晰明了，不易产生歧义。

20.5.2　系统界面操作流程

按照一般的操作习惯，本项目涉及的界面及界面间的关系，可以用图 20-26 表示。主要包括：界面外观和布局、用户操作流程、界面输入和输出、界面对应的逻辑操作等。

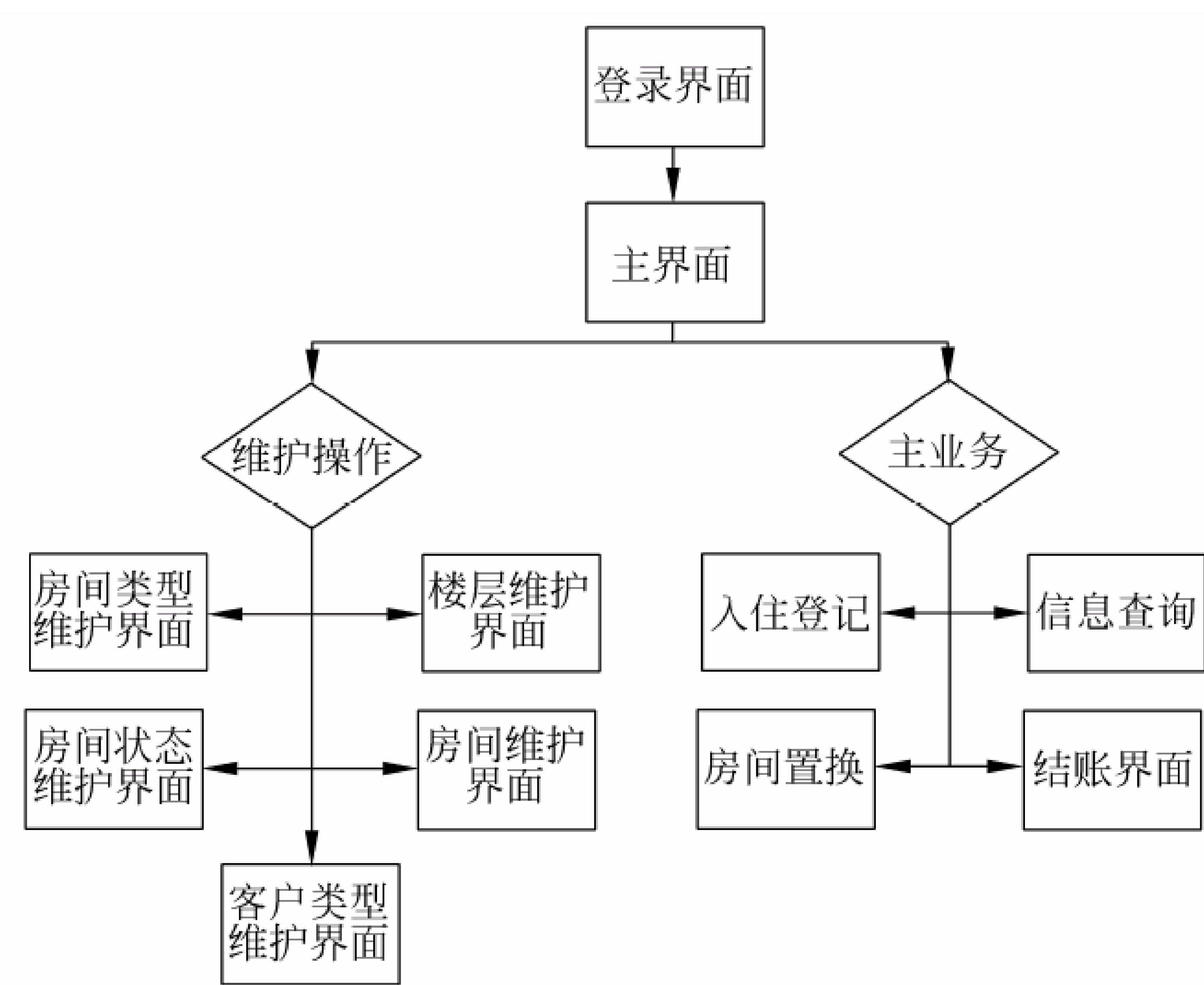


图 20-26　用户界面流程

由于宾馆信息管理软件业务本身不由界面决定，所以这里就不为每个界面进行外观和布局上的设计了，但需要更详细地定义每个界面的具体功能及它所需要显示或操作的数据。宾馆信息管理中，各个界面功能的具体定义如表 20-3 所示。

表 20-3　软件界面功能定义

界　　面	功　　能	相 关 数 据
登录界面	提供登录系统的功能，操作员输入用户名和密码，单击登录。登录失败提示，否则跳转到对应的操作界面	登录名输入框、密码输入框、登录按钮、退出按钮
操作主界面	为操作员提供各功能的进入菜单	包括菜单：系统维护，主业务
房间类型维护界面	只读形式显示当前已经维护过的房间类型信息。选中可以进行修改操作，也可以进行删除，但是已经在使用的类型不能删除	列表控件，文本输入框和相应的常规操作菜单
房间状态维护界面	由于房间状态属于稳定型数据，故不提供专属维护界面，可以在房间列表修改房间状态	
楼层维护界面	可以增加和修改删除楼层信息	常规菜单和列表控件，楼层名称文本控件
房间维护界面	以列表的形式显示已经维护好的房间信息，选中可以进行修改和删除操作，新增的时候需要选中类型和楼层及状态信息	常规操作菜单、下拉列表框和列表控件
客户类型维护界面	维护客户的级别和该级别对应的折扣比率	常规菜单和文本输入框及列表控件

续表

界　面	功　能	相关数据
入住登记界面	由于入住登记属于业务操作，此功能整合在房间列表，当房间状态为“空闲”时，双击即可录入入住登记信息	列表控件和客户基本信息输入控件以及确定按钮
房间置换界面	此功能相当于“结账”+“入住登记”	
结账界面	此功能整合在房间列表中，双击房间信息，当房间状态为“占用”时，就可以进行结账操作，显示结账界面和消费情况	只读形式显示客户信息文本控件
信息查询界面	查询宾馆管理决策的基本信息，如“入住率”，“收益”等	按日期区间选择日期控件、查询结果显示文本控件

20.6　宾馆管理软件的具体实现

在功能分析和数据库创建完备的情况下，就可以进入到编码阶段。前面已经说过，本系统采用三层架构，基于.NET 平台的 Win Form+ Ado.net 编程，使用程序语言描述上述业务需求，下面就各个重点功能模块结合代码进行详细技能点分析。

在此之前，需要确定本项目的文档规范，包含文件夹、代码放置和控件命名等规范，这也是一个项目开始要进行准备的重要文档之一。本例约定所有窗体均以 Frm_功能描述为名称，维护类界面放置在 maintenance 文件夹，业务类代码文件放置在 business 文件夹，其余变量及控件命名规范参考标准的微软建议规范。

20.6.1　系统主界面

系统主界面指的是登录后的集合所有功能菜单的控制界面。可以看成是“指挥控制中心”。常规系统主界面功能有菜单导航，权限控制和总体信息概览等。本例主要是系统导航和登录信息显示等功能，采取多文档技术，即 MDI 窗体实现，上面放置了一条 MenuStrip 控件，负责各个功能界面的导航。导航代码如示例代码 20-1 所示。

示例代码 20-1

```
private void tsmi user Click(object sender, EventArgs e)
        {
            //操作员维护界面
            Frm_user f = new Frm_user();
            f.MdiParent = this;   //表示新创建窗体的父窗体是 this 即本类
            f.Show();             //显示新创建的窗体
        }
```

同时还应该显示一些附件信息，如当前登录名，用于确定使用者的身份，还有显示当前时间等，都是一些能够提升客户体验的小功能。这里的时间要求能够动态刷新，需要用到 timer（定时器）控件。设置 timer 控件为可用，并设置循环执行的时间周期为 1 000（timer 的单位是毫秒），在它的唯一事件 Tick 下写入循环执行的代码，如示例代码 20-2 所示。

示例代码 20-2

```
private void tm info Tick(object sender, EventArgs e)
{
    //定时执行的代码，达到刷新的效果
    tssl_time.Text = DateTime.Now.ToString("yyyy年MM月dd日 HH:mm:ss");
}
```

20.6.2　系统登录窗体

登录窗体是任何信息管理类系统的第一个窗体。也是验证用户是否有权限使用系统的第一步，承担着整个系统的安全第一关责任。一般是根据用户提供的用户名和密码对数据库进行验证，通过则可以使用系统提供的服务，否则将拒绝用户所有操作。本例使用用户填写的用户名去数据库中查询原始密码，然后和用户输入的密码进行对比，得出验证结果，数据访问层（HotelUserService）查询代码如示例代码 20-2 所示。

示例代码 20-2

```
/// <summary>
/// 根据登录名查询用户信息
/// </summary>
public HotelUser GetUserByName(string name)
{
    try
    {
        return (from user in hotelDataContext.HotelUser
                where user.LoginName == name
                select user).SingleOrDefault();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

返回的是查询出的用户对象，在业务逻辑层（HotelUserManager）只需要取其密码属性对比即可。对比思路如示例代码 20-3 所示。

示例代码 20-3

```
/// <summary>
/// 根据用户名和密码验证账户合法性
/// </summary>
public bool IsLogin(string name, string pass)
{
    HotelUser user = hotelUserService.GetUserByName(name);
    if (user == null)
        return false;
    else if (pass == user.Password)
        return true;
    else
        return false;
}
```

🔔注意：上个代码片段里用到了 SingleOrDefault()方法，含义是如查询到，则返回第一条记录，如查询不到则返回空，不抛异常。

20.6.3　房间类型维护界面

本界面的功能在于系统启动时维护宾馆所有能够提供的房子类型信息，由于房间价格是根据类型决定的，所以本界面需要提供“类型名称”和“价格”信息的维护。为了运营方便，可以适当增加“备注”输入框。主要的业务是对上述信息进行增、删、改、查操作。但是千万注意，由于房间类型一旦被使用，就形成依赖关系，不可以再次删除。

本窗体在加载的时候就需要显示已经维护过的房间类型信息，所以在加载事件下查询所有的类型数据，数据访问层代码如示例代码 20-4 所示。

示例代码 20-4

```
/// <summary>
/// 获取所有房间类型信息
/// </summary>
public Table<RoomType> GetAll()
{
    try
    {
        return hotelDataContext.RoomType;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

业务逻辑层在这个方法中只起到一个数据传递的作用，这里不再赘述。表示层需要绑定列表控件，绑定代码如示例代码 20-5 所示。

示例代码 20-5

```
/// <summary>
/// 绑定 DataGridView
/// </summary>
private void BindList()
{
    roomTypeManager = new RoomTypeManager();
    dgv_list.DataSource = roomTypeManager.GetAll();
}
```

当单击列表中的任何一项时，就视为选中，需要把选中的数据在上面的编辑框中显示出来，以备修改或查看详细。所以需要给列表控件增加单击事件，取列表中的值显示到上面文本框中，代码如示例代码 20-6 所示。

示例代码 20-6

```
if (dgv_list.SelectedRows[0].Cells[0].Value != null)
{
```

```
    //获取选中行的第一行第 N+1 列的值，公式如 dgv list.SelectedRows[行][列].value
    txt price.Text = dgv list.SelectedRows[0].Cells[2]. Value. ToString();
    txt remark.Text = dgv list.SelectedRows[0].Cells[3]. Value. ToStr-
    ing();
    txt_typeName.Text = dgv_list.SelectedRows[0].Cells[1]. Value. ToString();
    op = OPMethod.Update;        //修改操作
    ChangeState();
}
```

其中，第一句是判断是否有选中行，如果不判断，则单击列表控件的空白部分就会出现异常。OP 是定义的一个枚举，用于表示当前是什么操作，ChangeState()方法用于控制界面的控件可用或不可用的状态。

当用户单击“新增”按钮时，所有文本框状态置为“可用”并且清空，用户填写完毕数据即可单击“保存”按钮进行信息保存。保存操作需要构建新的对象，表示层代码如示例代码 20-7 所示。

示例代码 20-7

```
if (op == OPMethod.Add)
{
    RoomType rt = new RoomType {  TypeName=txt_typeName.Text, TypePrice=
    Convert. ToDecimal(txt price.Text), Remark=txt remark.Text};
    if (roomTypeManager.AddRoomType(rt))
        MessageBox.Show("房间类型增加成功! ");
    else
        MessageBox.Show("类型名存在! ");
}
```

其中，创建对象用了 Visual Studio 2010 的特性，直接在创建时就给属性赋值，把构建好的对象传递到业务逻辑层，需要判断此房间类型名称是否存在，如果存在就不能添加，业务判断代码如示例代码 20-8 所示。

示例代码 20-8

```
/// <summary>
/// 增加房间类型
/// </summary>
 public bool AddRoomType(RoomType rt)
 {
     if (roomTypeService.GetRoomTypeByTypeName(rt.TypeName) != null)
     {
         return false;
     }
     else
     {
         roomTypeService.AddRoomType(rt);
         return true;
     }
 }
```

如果不存在重复的房间类型名称，则进行新增操作，数据访问层插入代码如示例代码 20-9 所示。

示例代码 20-9

```
/// <summary>
/// 增加房间类型
/// </summary>
public void AddRoomType(RoomType rt)
{
    try
    {
        hotelDataContext.RoomType.InsertOnSubmit(rt);
        hotelDataContext.SubmitChanges();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

以上是所有保存业务涉及的代码。下个业务是修改，修改的原理一样，也需要在表示层构建好新的对象，把要修改的新值赋给该对象的属性，然后传递给业务逻辑层，代码如示例代码 20-10 所示。

示例代码 20-10

```
else if (op == OPMethod.Update)
{
    int typeId = int.Parse(dgv_list.SelectedRows[0].Cells[0].
    Value. ToString());
    RoomType rt = new RoomType {  Remark=txt_remark.Text, TypeName=txt_
    typeName.Text, TypePrice=Convert.ToDecimal(txt_price. Text), TypeID=
    typeId};
    if(roomTypeManager.UpdateRoomType(rt))
    MessageBox.Show("修改成功！");
    else
        MessageBox.Show("修改失败！");
}
```

业务逻辑层负责数据的传递，而数据访问层才是真正的数据操作。LINQ 修改操作有一个约定就是先把要修改的实体查询出来，然后赋新值，提交保存。代码如示例代码 20-11 所示。

示例代码 20-11

```
/// <summary>
/// 修改房间类型信息
/// </summary>
public void UpdateRoomType(RoomType roomtype)
{
    try
    {
        RoomType oldType = (from rt in hotelDataContext.RoomType
                            where rt.TypeID == roomtype.TypeID
                              select rt).Single();//选择符合 ID 的记录
        oldType.TypeName = roomtype.TypeName;
        oldType.TypePrice = roomtype.TypePrice;
        oldType.Remark = roomtype.Remark;
         hotelDataContext.SubmitChanges();
```

```
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

删除业务相对简单，只需要把主键字段传递到数据访问层，查询出要删除的实体，调用删除方法即可完成。代码如示例代码 20-12 所示。

示例代码 20-12

```
/// <summary>
/// 根据 ID 删除类型信息
/// </summary>
public void DelRoomType(int typeId)
{
    try
    {
        hotelDataContext.RoomType.DeleteOnSubmit((from rt in hotelDataCon-
        text.RoomType
                  where rt.TypeID==typeId
                  select rt).Single());
         hotelDataContext.SubmitChanges();//提交修改
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

声明：界面用户输入需要有数据合法性验证，所有的操作都必须有结果提示，这样才方便用户使用。

20.6.4　房间信息维护界面

房间信息是指整个宾馆所有的可用房间情况，包含的附加维护信息也比较多，如楼层、类型和状态等，最后需要对特殊的房间设施或物品进行一些附加备注说明，涉及的操作还是增、删、改和查。

和上个业务一样，在界面加载时就需要把列表信息显示出来，同时还要把如楼层等的信息以下拉列表框的方式展示出来，以备选择。需要注意的是，列表控件显示的只是必要信息，也不全来自一张表，所以显示的难度就大些。根据 LINQ 技术，可以在界面上再次进行筛选，只拿出需要的数据，以匿名类的方式返回，代码如示例代码 20-13 所示。

示例代码 20-13

```
/// <summary>
/// 绑定房间 datagridview
/// </summary>
private void BindDgv()
{
    //绑定房间列表
    RoomManager roomManager = new RoomManager();
```

```
    Table<Room> rs = roomManager.GetRoomInfo();
    dgv list.DataSource = (from r in rs
        select new { Number = r.Number, TypeName = r.RoomType.TypeName,
        Description = r.Description, FloorName = r.Floor.FloorName,
        StateName = r.RoomState.StateName, RoomId = r.RoomID }).ToList();
}
```

绑定完毕列表，还需要绑定楼层和类型信息。思路与后面一样，先指定数据源，然后设置“显式值”和“隐式值”，用户选择的是显示值，后台可以获取到隐式值。代码如示例代码 20-14 所示。

示例代码 20-14

```
/// <summary>
/// 绑定下拉列表框
/// </summary>
private void BindList()
{
   //绑定房间类型
   RoomTypeManager roomTypeManager = new RoomTypeManager();
   cbo type.DataSource = roomTypeManager.GetAll();
   cbo type.DisplayMember = "TypeName";    //显式值
   cbo_type.ValueMember = "TypeID";        //隐式值

   //绑定房间状态
   …//省略类似代码
   //绑定楼层信息
   …//省略类似代码
}
```

为了防止用户随意输入楼层信息的值，需要把 combox 的 DropDownStyle 属性设置为 DropDownList，这样用户就只能选择而不能输入，避免了不必要的输入风险。接下来是保存按钮的功能，由于新增和编辑都需要保存，所以把两个业务放在一个按钮下完成。完成的思路和上个业务类似，都需要构建好实体，传递给业务逻辑层，都需要验证名称的合法性。从本段代码之后，类似功能的代码就不再列出了。代码如示例代码 20-15 所示。

示例代码 20-15

```
private void tsb_save_Click(object sender, EventArgs e)
{
   if (op == OPMethod.Add)
   {
      Room r = new Room { Number = txt roomNo.Text, RoomStateId=Convert.
      ToInt32(cbo state.SelectedValue),  TypeID=Convert.ToInt32(cbo
      type. SelectedValue), FloorId=Convert.ToInt32(cbo floor. Selected-
      Value), Description=txt remark.Text };

            if (roomManager.AddRoom(r))
               MessageBox.Show("新增成功！");
            else
               MessageBox.Show("新增失败！");
         }
         else if (op == OPMethod.Update)
         {
            int roomId = Convert.ToInt32(txt roomNo.Tag);
            Room r = new Room { Number = txt_roomNo.Text, RoomStateId =
```

```
                Convert. ToInt32(cbo state.SelectedValue), TypeID = Convert.
                ToInt32 (cbo type.SelectedValue), FloorId = Convert.ToInt32
                (cbo floor.SelectedValue), Description = txt remark.Text,
                RoomID=roomId };
                if (roomManager.UpdateRoomInfo(r))
                    MessageBox.Show("修改成功! ");
                else
                    MessageBox.Show("修改失败! ");
            }
            op = OPMethod.Cancel;
            ChangeState();              //改变状态
            BindDgv();                  //绑定列表
        }
```

20.6.5　入住登记

入住登记属于本项目中重点功能之一。入住登记的第一步是判断某个指定的房间状态是否为空闲，如果是空闲则允许登记，否则给出提示。本例采用的思路是以图标的形式显示所有客房状态信息，服务员用眼睛就可以非常直观地看到哪些房间是可以登记的，一旦出现手误，系统会有相应的提示信息。绑定房间信息为图标使用的是 ListView 控件的大图标模式，要想显示图标就必须提前准备一套图标，绑定代码如示例代码 20-16 所示。

示例代码 20-16

```
/// <summary>
/// 绑定房间图标列表
/// </summary>
private void ShowList()
{
    lv room.Items.Clear();
    RoomManager roomManager = new RoomManager();
    List<Room> rs = roomManager.GetRoomInfo().ToList();
    int picIndex = 0;
    for (int i = 0; i < rs.Count; i++)
    {
        lv room.Items.Add(rs[i].Number);
        switch (rs[i].RoomState.StateName)
        {
            case "空闲": picIndex = 2; break;
            case "占用": picIndex = 4; break;
            case "维修": picIndex = 3; break;
            case "清扫": picIndex = 1; break;
            case "预定": picIndex = 5; break;
        }
        lv room.Items[i].ImageIndex = picIndex;
        lv room.Items[i].Tag = rs[i].RoomState.StateName;
        lv room.Items[i].Name = rs[i].RoomID.ToString();
    }
}
```

双击状态为“空闲”的房间图标，弹出客户信息录入界面，判断代码如示例代码 20-17 所示。

示例代码 20-17

```
private void lv room MouseDoubleClick(object sender, MouseEventArgs
e)
{
    if (lv room.SelectedItems.Count > 0)
    {
        if (lv_room.SelectedItems[0].Tag.ToString() == "占用")
        {
            Frm_chargeDetails f = new Frm_chargeDetails(int. Parse
            (lv_room.SelectedItems[0].Name));
            f.ShowDialog();
            ShowList();
        }
        else if (lv_room.SelectedItems[0].Tag.ToString() != "空闲")
        {
            MessageBox.Show("房间状态为" + lv_room.SelectedItems[0]. Tag.
            ToString() + ",请确定! ","温馨提示",MessageBoxButtons. OK,Message-
            BoxIcon.Information);
        }
        else
        {
            //弹出客户信息录入界面，并传递 2 个参数
            Frm_GuestInfo f = new Frm_GuestInfo(int. Parse(lv_room.
            SelectedItems[0].Name), lv_room.SelectedItems[0].Text);
            f.ShowDialog();
            ShowList();//最后刷新全部房间状态
        }
    }
}
```

弹出的客户信息登记界面同样需要绑定客户类型信息，前面已经讲过，这里不再赘述。在录入客户信息时，由于为了促进客户消费，对客户建立消费档案以便享受更多的优惠，所以在录入信息时需要判断是否是老客户。判断的依据是：“客户名称”和“证件号码”同时匹配就算是相同客户，系统自动查询出该客户的信息，不需要继续录入，本次的消费会累加。信息完善之后记入明细账，客户开房成功。

判断客户是否存在，是在证件号码输入框的光标离开事件下完成，实现的代码如示例代码 20-18 所示。

示例代码 20-18

```
private void txt pid Leave(object sender, EventArgs e)
{
    //这里根据客户名称和证件号码，检索此客户以前是否来过
    GuestInfo guest = guestInfoManager.GetInfoByNameAndPid(txt_name.Text,
    txt_pid.Text);
    if (guest != null)
    {
        lbl_msg.Text = "该客户信息已经存在 ! ";
        txt tel.Text = guest.Mobile;
        cbo type.Text = guest.GuestCategory.CategoryName;
        rdo_male.Checked = guest.Sex == "男" ? true : false;
        guestId = guest.GuestID;
    }
}
```

客户信息录入完成之后，需要进行记账和修改房间状态操作。记账需要区分新老客户，对于老客户，只需要记入明细账改变房间状态即可，对于新客户还要插入客户信息。下面以新客户为例说明三层之间的操作，表示层需要构建两个实体，一个是客户信息，一个是明细账，实现代码如示例代码 20-19 所示。

示例代码 20-19

```
//构建客户实体
GuestInfo guest = new GuestInfo { ChargeSum = 0, CategoryID=Convert. ToInt
32(cbo type.SelectedValue),   Mobile=txt tel.Text,   Name=txt name.Text,
PID=txt_pid.Text, Sex=rdo_male.Checked?"男":"女"};
//构建明细记录部分属性
DetailRecord dr = new DetailRecord { Charge = 0, InTime = DateTime.Now, RoomID
= roomId };
RoomManager roomManager = new RoomManager();
roomManager.OpenRoomNewGuest(roomId, guest, dr);
```

对应的业务逻辑层就复杂些了，需要进行插入客户信息、插入明细账、修改房间状态信息等操作，实现代码如示例代码 20-20 所示。

示例代码 20-20

```
public void OpenRoomNewGuest(int roomId,GuestInfo guest,DetailRecord dr)
{
    //1. 插入客户信息表
   GuestInfoService guestInfoService = new GuestInfoService();
   int guestId= guestInfoService.AddGuest(guest);
   //2. 插入明细表
   DetailRecordService DetailRecordService = new DetailRecordService();
   dr.GuestID = guestId;
   DetailRecordService.AddDetailRecord(dr);
   //3. 更新房间表状态字段
   Room r = new Room { RoomID=roomId, RoomStateId=3 };
   roomService.UpdateRoomState(r);
}
```

上述代码分别调用了 3 个数据访问层类的 3 个方法，这里暂时没有考虑事务。其中修改房间状态代码是通用的，只需要构建一个包含新状态和 ID 的实体传递过去即可，参考代码如示例代码 20-21 所示。

示例代码 20-21

```
/// <summary>
/// 修改房间状态为
/// </summary>
public void UpdateRoomState(Room room)
{
   try
   {
      Room oldRoom = (from r in hotelDataContext.Room
                      where r.RoomID == room.RoomID
                      select r).Single();
      oldRoom.RoomStateId = room.RoomStateId;
      hotelDataContext.SubmitChanges();
   }
```

```
    catch (Exception ex)
    {
        throw ex;
    }
}
```

20.6.6　结账界面

结账是在客户离开宾馆的时候发生，也是重点业务之一。回到房间列表界面找到需要结账的房间图标，系统会自动判断该房间是否满足结账要求，如果满足会弹出结账界面，如果不满足则会给出相应的提示，具体的判断代码前面已经给过。这里只需要讨论下结账界面的功能。

结账界面加载之后需要显示客户信息进行核对，同时需要显示房间信息和客户进行核对，将所有文本框置为只读，最后根据入住时间和房间类型计算出对应的消费金额，即客户应付账款。计算公式不难，就是天数×单价，这里的问题在于求出客户入住天数，用到一个求天数差方法，该方法详细代码如示例代码 20-22 所示。

示例代码 20-22

```
/// <summary>
/// 计算日期间隔
/// </summary>
/// <param name="d1">要参与计算的其中一个日期</param>
/// <param name="d2">要参与计算的另一个日期</param>
/// <param name="drf">决定返回值形式的枚举</param>
/// <returns>一个代表年月日的 int 数组，具体数组长度与枚举参数 drf 有关</returns>
public static int[] toResult(DateTime d1, DateTime d2, diffResultFormat drf)
{
    #region 数据初始化
    DateTime max;
    DateTime min;
    int year;                    //定义年
    int month;                   //定义月
    int tempYear, tempMonth;     //定义临时变量
    if (d1 > d2)
    {
        max = d1;
        min = d2;
    }
    else
    {
        max = d2;
        min = d1;
    }
    tempYear = max.Year;
    tempMonth = max.Month;
    if (max.Month < min.Month)
    {
        tempYear--;
        tempMonth = tempMonth + 12;
    }
    year = tempYear - min.Year;
    month = tempMonth - min.Month;
```

```
    #endregion
    #region 按条件计算
    if (drf == diffResultFormat.dd)
    {
        TimeSpan ts = max - min;
        return new int[] { ts.Days };
    }
    if (drf == diffResultFormat.mm)
    {
        return new int[] { month + year * 12 };
    }
    if (drf == diffResultFormat.yy)
    {
        return new int[] { year };
    }
    return new int[] { year, month };
    #endregion
}
```

其中 diffResultFormat 是一个枚举，枚举的定义如示例代码 20-23 所示。

示例代码 20-23

```
/// <summary>
/// 关于返回值形式的枚举
/// </summary>
public enum diffResultFormat
{
    /// <summary>
    /// 年数和月数
    /// </summary>
    yymm,
    /// <summary>
    /// 年数
    /// </summary>
    yy,
    /// <summary>
    /// 月数
    /// </summary>
    mm,
    /// <summary>
    /// 天数
    /// </summary>
    dd,
}
```

单击结账按钮时要执行一系列操作，如更新房间状态，更新客户消费，更新明细账离开时间等。对应的业务逻辑层代码如示例代码 20-24 所示。

示例代码 20-24

```
/// <summary>
/// 结账操作
/// </summary>
public void Charge(int roomId, int detailId, int guestId,decimal money)
{
    //1.更新明细表的离开时间字段和消费
    DetailRecordService detailRecordService = new DetailRecordService();
    detailRecordService.UpdateLeaveTimeAndMoney(detailId,money);
```

```
    //2. 更新客户的消费字段
    GuestInfoService guestInfoService = new GuestInfoService();
    guestInfoService.UpdateGuestMoney(guestId, money);
    //3. 更新房间状态为清扫
    RoomService roomService = new RoomService();
    Room r = new Room { RoomID=roomId, RoomStateId=2 };
    roomService.UpdateRoomState(r);
}
```

其中数据访问层的代码这里省略，和上面的操作基本一致。

20.6.7　查询功能

作为宾馆的管理层必须对宾馆的经营状况负责，该宾馆的经营数据必须实时掌握。可以自主选择查询时间区间，默认是一个月，采用的是日历控件，初始化时间段代码如示例代码 20-25 所示。

示例代码 20-25

```
private void Frm_accountInfo_Load(object sender, EventArgs e)
{
    time_start.Value = DateTime.Now.AddMonths(-1);//日期前推一个月
    time_end.Value = DateTime.Now;//终止时间为当前时间
}
```

酒店宾馆行业里面说的最多的，也是大家最关心的无非就是“入住率”这个指标了。入住率又叫客房出租率，是指一段时期实际出租客房间（天）数占这段时期可出租客房间（天）的百分比。

$$即本期客房出租率=\frac{本期实际出租客房（天）数}{饭店可出租客房总数\times本期天数}\times100\%$$

例如丽园饭店是拥有 300 间客房的中档饭店，2007 年该饭店出租的客房（天）数为 76 650 间天，那么丽园饭店 2007 年客房出租率是：

$$客房出租率=\frac{76\ 650间天}{300间\times365天}\times100\%=70\%$$

这里需要计算出天数和实际出租客房天数，根据选择的时间段查询该时间段所有明细账，累加天数即为实际出租天数，在业务逻辑层实现，实现思路如示例代码 20-26 所示。

示例代码 20-26

```
/// <summary>
/// 根据时间段获取入住天数
/// </summary>
public int GetUseDaysByTime(DateTime start, DateTime end)
{
    List<DetailRecord> rs = detailRecordService.GetDetailByTime (start,
    end);
    int days = 0;
    foreach (DetailRecord item in rs)
    {
```

```
        days += Common.toResult((DateTime)item.InTime, (DateTime)item. Lea-
        veTime, diffResultFormat.dd)[0];
    }
    return days;
}
```

然后表示层的计算和实现方式就相对简单了，代码如示例代码 20-27 所示。

示例代码 20-27

```
int totalRoomNum = 0;        //房间总数
int factUseDays = 0;         //实际出租天数
int days = 0;                //本期查询区间天数
private void btn_inRate_Click(object sender, EventArgs e)
{
    RoomManager roomManager = new RoomManager();
    totalRoomNum = roomManager.GetRoomInfo().Count();
    DetailRecordManager detailRecordManager = new DetailRecordManager();
    factUseDays = detailRecordManager.GetUseDaysByTime(time_start.Value,
    time_end. Value);
    days=Common.toResult(time_start.Value,time_end.Value,diffResult-
    Format.dd)[0];
    decimal rateInUse = (decimal)factUseDays / (totalRoomNum * days);
    txt_rateInUse.Text = (rateInUse * 100).ToString().Substring(0,5) + "%";
    txt_income.Text = detailRecordManager.GetInComeByTime(time_start.
    Value, time_end.Value).ToString();
    dgv_list.AutoGenerateColumns = false;//设置自动产生列为否
    List<DetailRecord> drs =detailRecordManager.GetDetailByTime(time_
    start. Value, time_end.Value);
    dgv_list.DataSource = (from d in drs select new { RecordID = d.RecordID,
    Name = d.GuestInfo.Name, Number = d.Room.Number, InTime = d.InTime,
    LeaveTime = d.LeaveTime, Charge = d.Charge }).ToList() ;
}
```

把计算结果显示出来即可。

这里需要强调一点的是，在使用 LINQ 查询时除了上述传统做法之外还可以采用“Lambda 表达式”，它是一个匿名函数，可以包含表达式和语句，并且可用于创建委托或表达式目录树类型。详细介绍请参考前面章节，具体使用方式举例如示例代码 20-28 所示。

示例代码 20-28

```
/// <summary>
/// 根据时间段获取入住信息
/// </summary>
public List<DetailRecord> GetDetailByTime(DateTime start, DateTime end)
{
    try
    {
        return hotelDataContext.DetailRecord.Where(d=>d.InTime>=start&&d.
        LeaveTime<=end).ToList();
    }
    catch (Exception ex)//捕获异常
    {
        throw ex;
    }
}
```

最后一个查询业务就是客户入住历史查询了，这个查询中列表的显示不是重点，重点在于“入住次数”和“总花费”的计算。入住次数计算思路是统计明细账中的记录条数即可，使用下面代码实现，如示例代码 20-29 所示。

示例代码 20-29

```
DetailRecordManager detailRecordManager = new DetailRecordManager();
List<DetailRecord> drs=detailRecordManager.GetDetailByTimeAndName(time_
start.Value, time_end.Value, txt_name.Text);
dgv_list.DataSource = (from d in drs select new { RecordID = d.RecordID,
Name = d.GuestInfo.Name, Number = d.Room.Number, InTime = d.InTime, LeaveTime
= d.LeaveTime, Charge = d.Charge }).ToList();
txt_totalMoney.Text = drs.Sum(d=>d.Charge).ToString();//对指定列求和
txt_UseTimes.Text = drs.Count().ToString();            //求集合数量
```

同样，为了显示必要的数据，这里也用到了“匿名类”。

20.7　项 目 演 示

本项目案例整个开发过程暂时告一段落，在完成了项目测试后正式上线前需要有一个项目演示培训的过程，这个过程偏重于讲解流程和功能及操作，具体实现暂不涉及。启动软件看到的第一个界面就是登录界面，如图 20-27 所示，默认的用户名和密码都是 admin，输入后单击“登录”按钮即可看到主操作窗体。

登录成功后显示主界面，在其中可以进行基础信息的维护操作和业务及查询操作，如图 20-28 就是维护房间信息的界面。



图 20-27　登录界面



图 20-28　房间信息维护

在基本信息维护完毕之后就可以正式投入运营了，进行关键业务，比如开房操作。本案例采用的是图示方式操作，简单方便。如图 20-29 就是房间信息列表界面。



图 20-29　房间信息列表

可以根据客户的需要　选客户需要的房间进行开房操作，开房时需要录入客户信息，客户信息如有存在则自动查询并显示，整个操作流程如图 20-30 所示。



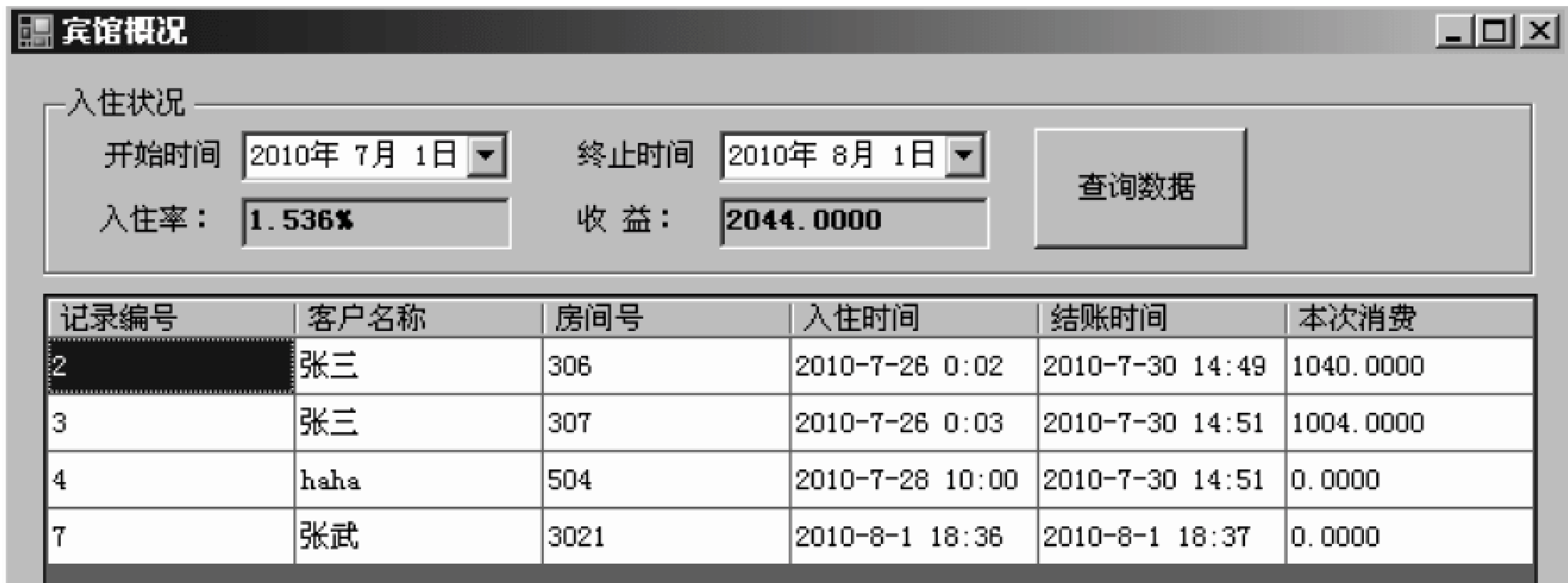
图 20-30　开房录入客户信息界面

如果需要结账，则找到客户入住的房间双击图标，系统会自动弹出结账窗体，同时显示客户和房间信息，可以进行结账操作，如图 20-31 所示。



图 20-31　结账界面

最后，作为宾馆管理层需要随时掌握宾馆运营情况，比如入住率等信息，可以通过查询业务实现，也可以根据需要查询任意时间段的数据，操作如图 20-32 所示。



记录编号	客户名称	房间号	入住时间	结账时间	本次消费
2	张三	306	2010-7-26 0:02	2010-7-30 14:49	1040.0000
3	张三	307	2010-7-26 0:03	2010-7-30 14:51	1004.0000
4	haha	504	2010-7-28 10:00	2010-7-30 14:51	0.0000
7	张武	3021	2010-8-1 18:36	2010-8-1 18:37	0.0000

图 20-32　查询界面

20.8　项 目 小 结

二十年之后，业界在面向对象（OO）编程技术的发展过程中趋于稳定。现在，程序员已经认为诸如类、对象和方法等特性是理所当然的。在探究当前的和下一代技术时，明显可以看出，有关编程技术的下一个难题是降低访问和集成特定信息（这些信息不是使用 OO 技术进行原始定义的）的复杂性。非 OO 信息的两个最常见源是关系数据库和 XML。本例使用 MSSQL 作为数据源，从三层搭建到实体映射，完整地操作一遍，整个项目案例中既用到当前最为流行的开发模式又用到了未来会流行的开发技术。但是作为项目本身来说，不完整，很多功能只是完成一个大致的模拟而已，希望能够起到抛　引玉的作用，助读者在开发领域更上一层楼。

第 21 章　XML 实例——ME 校友录

在本章的 XML 相关学习任务完成后，需要以一个实际项目的方式加以巩固，以便扎实、灵活地掌握本章所学技能点。本次项目案例是开发一个校友录的管理系统，从系统目标和需求分析开始，到系统框架的搭建和任务的分解，直至编码测试的完成。重点介绍校友录管理系统的实现过程：包括系统需求分析、系统调查、流程分析、数据流程分析、功能设计、数据库设计、系统物理配置方案、系统实现、系统测试和调试。本系统主要功能有班级成员注册、班级成员登录、班级成员信息维护、管理员设置、信息查看、注销等内容。

21.1　系 统 概 述

本校友录系统主要提供校友之间的信息交流，以网络为媒介，故采用的是 B/S 结构。前台使用 ASP.NET 技术框架，并通过 IIS 进行发布，后台采用 XML 作为轻量级数据库，实现校友录的基本功能。

21.1.1　功能概述

由于本校友录是基于 B/S 的网络应用系统，所面向的客户是学生，所要实现的功能是信息的共享，故应该实现下列功能：

- 学生注册。
- 学生登录。
- 自己和同学的信息查看。
- 管理员管理和设置。
- 个人信息修改。

21.1.2　可行性分析

由于本系统采用的是 XML 作为数据库，重点在于分析 XML 数据库的可行性。相比较传统关系型数据库，XML 数据库有以下优点：

- XML 数据库能够对半结构化数据进行有效的存取和管理。如网页内容就是一种半结构化数据，而传统的关系数据库对于类似网页内容这类半结构化数据无法进行有效的管理。

- 提供对标签和路径的操作。传统数据库语言允许对数据元素的值进行操作，不能对元素名称操作，半结构化数据库提供了对标签名称的操作，还包括了对路径的操作。
- 当数据本身具有层次特征时，由于 XML 数据格式能够清晰表达数据的层次特征，因此 XML 数据库便于对层次化的数据进行操作。XML 数据库适合管理复杂数据结构的数据集，如果已经以 XML 格式存储信息，则 XML 数据库利于文档存储和检索；可以用方便实用的方式检索文档，并能够提供高质量的全文搜索引擎。另外 XML 数据库能够存储和查询异种的文档结构，提供对异种信息存取的支持。

21.2　项目需求分析

和任何一个成功的项目一样，成功的需求是成功软件的根本。需求的确定，可以通过网络调查、问卷调查和面对面沟通等方式进行。确定需求开发过程，确定如何组织需求的收集、分析、细化并核实的步骤，并将它编写成文档。对重要的步骤要给予一定指导，这将有助于分析人员的工作，而且也使收集需求活动的安排和进度计划更容易进行。这个阶段，一切重点都是为了清楚用户需要什么，每个用户类别只派一位代表参与是不充分的，打组合拳，不要单一套路，在可能的情况下先小规模试验。记录数据同样重要，如果得到的需求信息太多，可以采取以下步骤：

- 让用户列出他们希望通过产品完成的各种目标。
- 让他们笼统列出达到目标的各种需求。
- 按自己的理解，把这些需求转述给用户，让他们判断是否正确。
- 鼓励用户以更开放的思路提出更多需求。
- 提供几个需求类别，让用户将所有需求“对号入座”。
- 列出各项需求的衡量标准，对不易衡量的，修改或删除。

由于本例是学习讨论用，故只提供现实的校友录最基本的需求，经过分析得出的总流程图如图 21-1 所示。

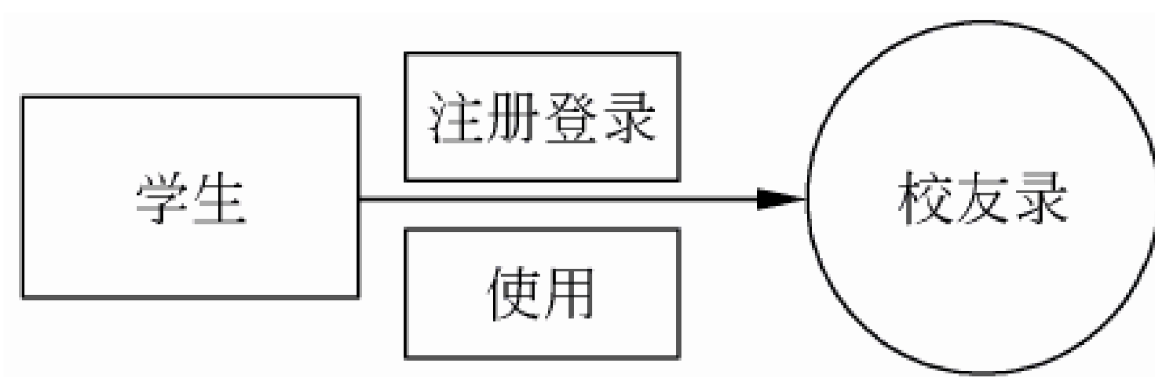


图 21-1　系统总流程图

21.2.1　系统功能分析

本校友录系统使用者按角色分为三类人：注册用户，班级管理员，系统管理员。功能比较简单，主要提供注册和信息查看功能，其业务描述如表 21-1 所示。

表 21-1　校友录系统功能说明

功 能 类 别	功能名称、标识符	描　　述
普通用户注册	用户注册	普通用户要想成为校友的一员，需要先进行个人信息录入
普通用户登录	用户登录	普通用户在进行注册操作之后，使用系统功能前需要进行的身份验证操作
个人信息查看	信息查看	可以根据需要查看自己的信息或者校友的信息
个人信息修改	信息修改	用户可以对自己的信息包括头像进行修改操作
搜索功能	校友搜索	根据关键字对校友信息进行搜索
管理员操作	管理登录	系统管理员的身份验证操作
	管理员设置	系统管理员可以设置普通管理员
注销	退出操作	清理登录状态

21.2.2　系统总用例分析

用例是指系统提供的业务功能与参与者的交互，表现问题领域中各实体间的联系和业务往来活动。它用于建立问题领域的业务用例模型，实现了一个系统中各个功能模块之间的分割，简化了开发任务，本系统的参与者如图 21-2 所示。

根据图 21-2 所示的参与者和上节分析的功能，可以画出整个系统的用例图，用来描述系统整体功能，如图 21-3 所示。

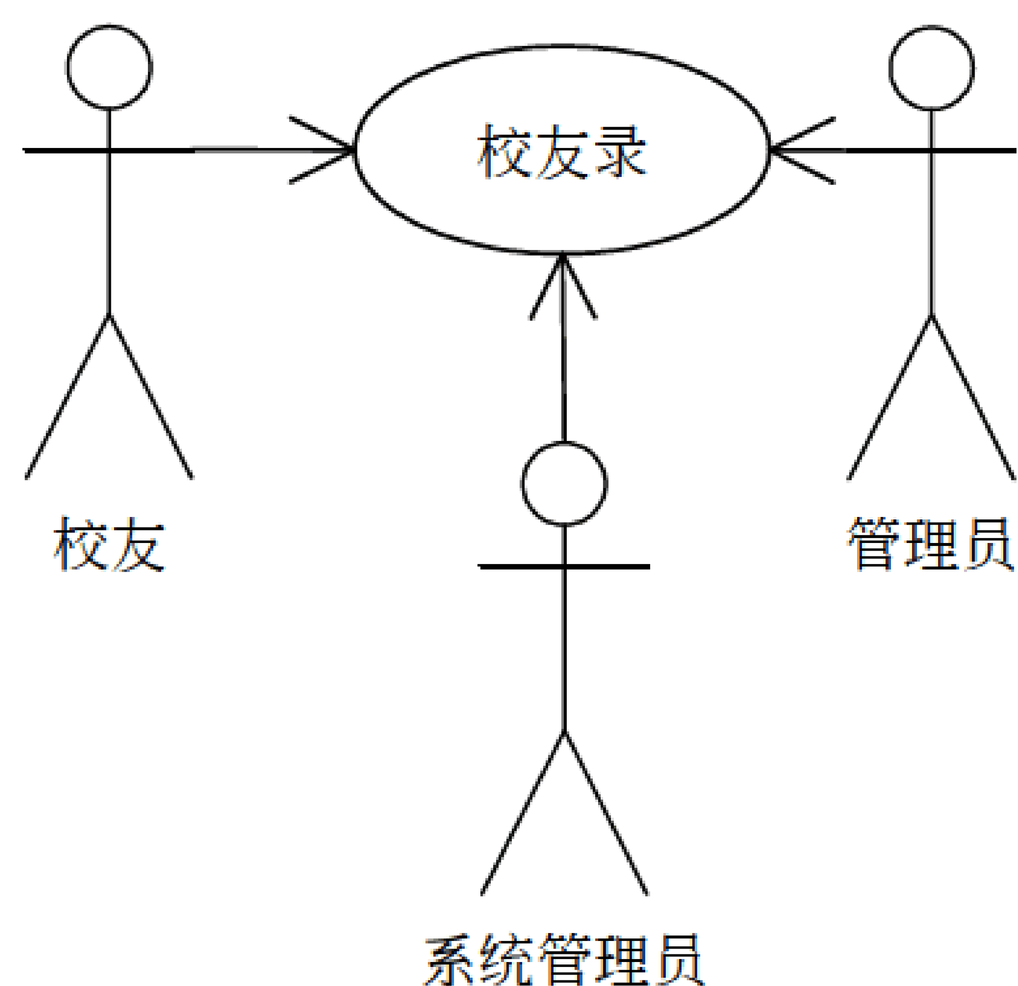


图 21-2　系统参与者模型

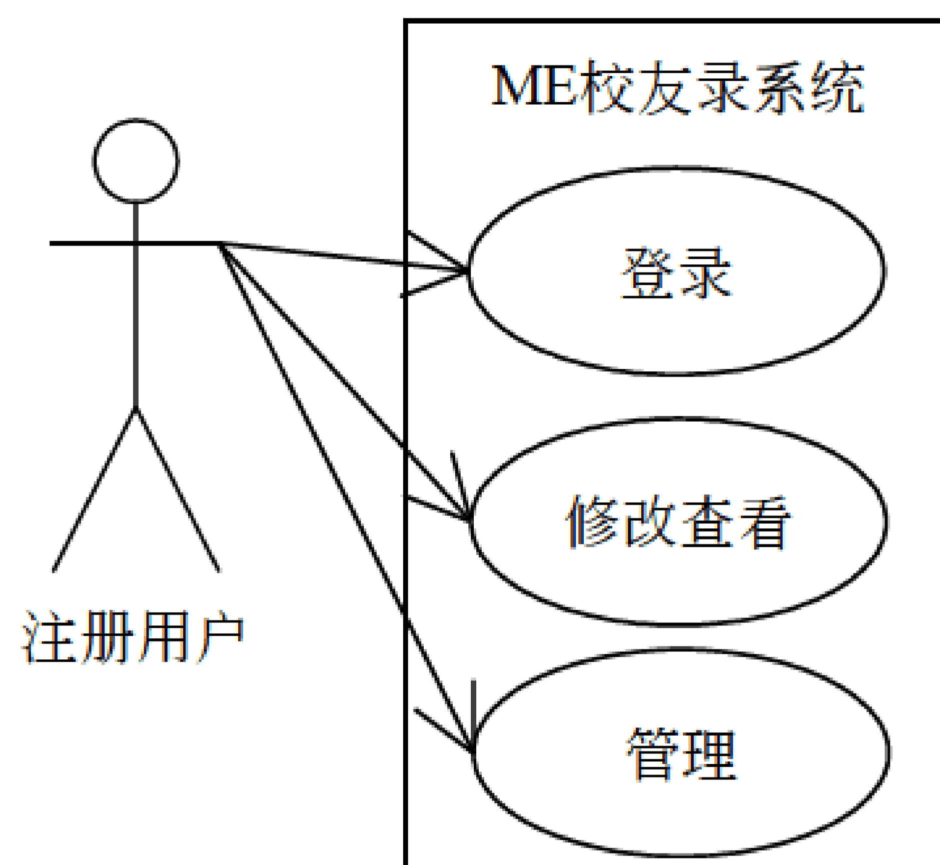


图 21-3　系统总用例图

21.2.3　系统用例分析

【用例 1：用户注册】

- ❑ 描述：要成为校友录的成员和大家共享信息，必须先注册，填写自己的信息。
- ❑ 参与者：用户。
- ❑ 用例图：图 21-4。

【用例 2：用户登录注销】

- ❑ 描述：为了验证客户身份必须经过登录操作，确认使用者的身份用。在不需要使用

系统功能的时候要进行注销，以避免秘密泄露。

- 参与者：用户。
- 用例图：图 21-5。

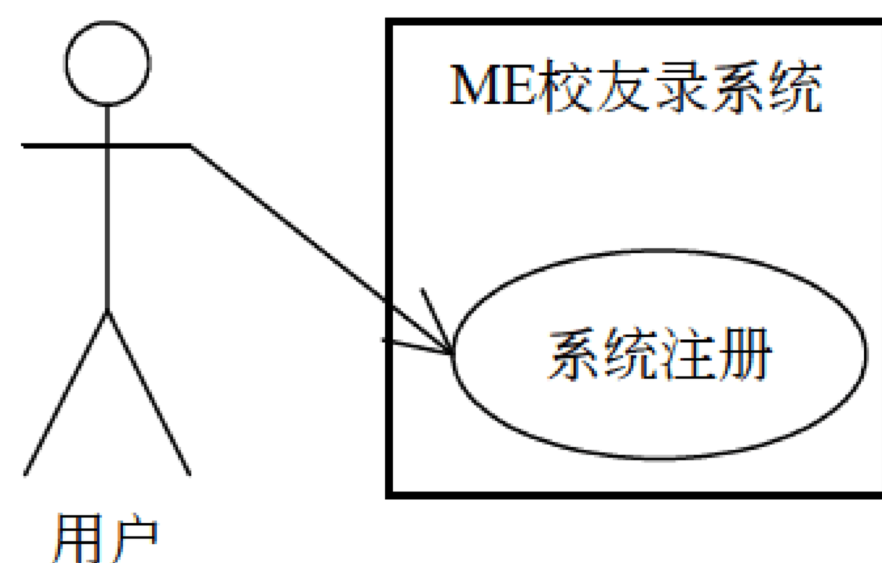


图 21-4　注册用例

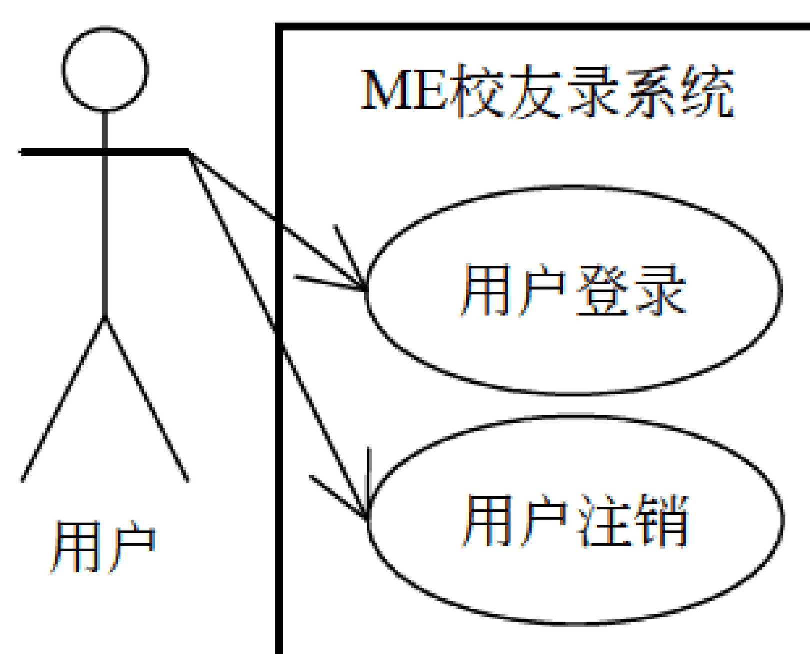


图 21-5　登录用例

【用例 3：用户信息查看】

- 描述：登录成功后可以查看和修改自己的信息（可以上传头像）。
- 参与者：用户。
- 用例图：图 21-6。

【用例 4：维护成员】

- 描述：系统管理员登录后，可以设置普通管理员，也可以删除成员（同时删除头像图片）。
- 参与者：管理员。
- 用例图：图 21-7。

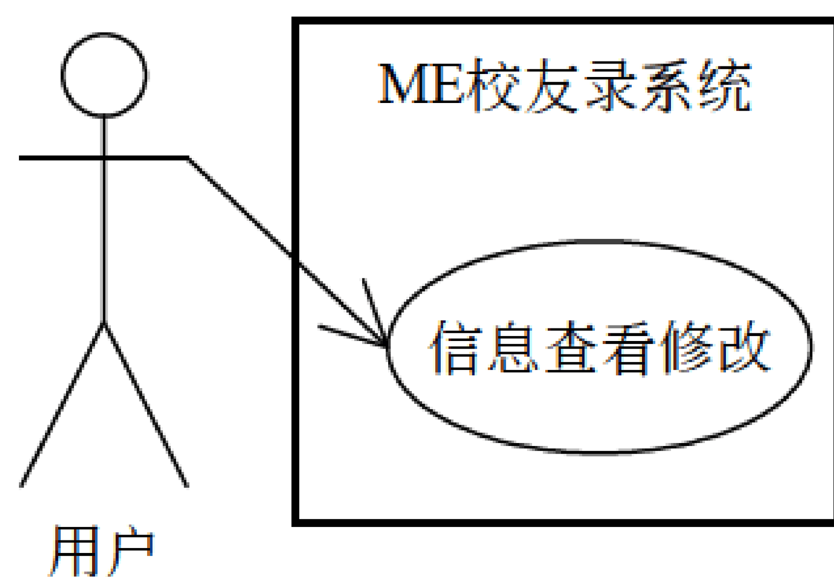


图 21-6　查看修改信息用例

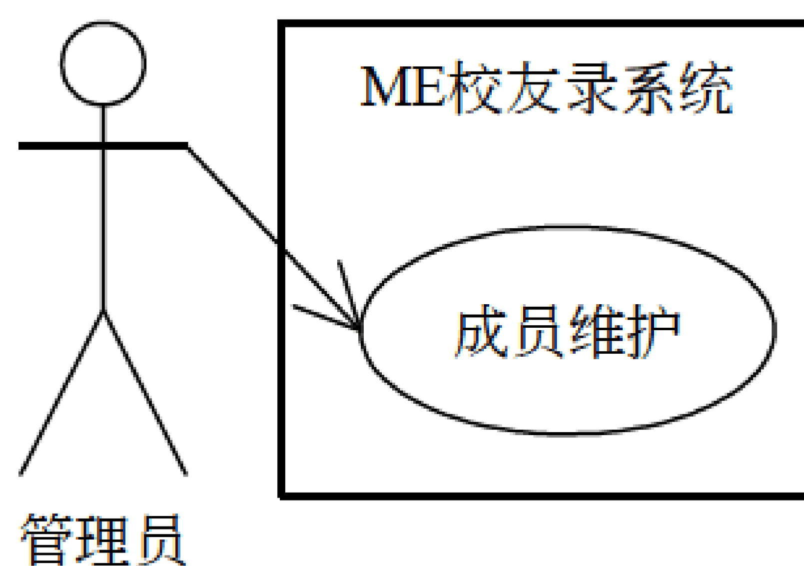


图 21-7　成员维护

【用例 5：成员搜索】

描述：已登录用户可以自定义关键字对校友信息进行搜索，结果以列表的形式显示。

- 参与者：管理员。
- 用例图：图 21-8。

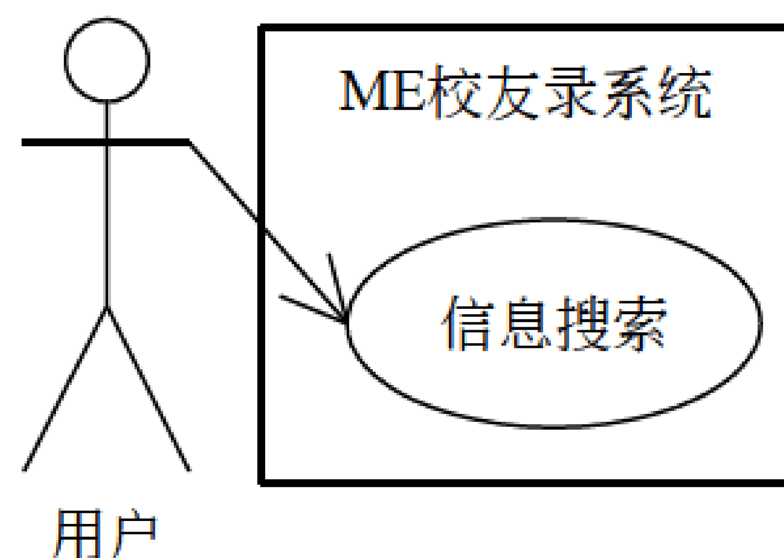


图 21-8　搜索用例

21.3　系统总体架构设计

B/S（浏览器/Web 服务器）结构是在 TCP/IP 的支持下，以 HTTP 为传输协议，客户端

通过 Browser 向 Web 服务器发送请求，Web 服务器部署 Web 引用程序，将程序的结果以 HTML 形式返回客户端浏览器的一种应用模式。在这种结构下用户工作界面是通过浏览器来展现的，最大的好处是用户计算机只需要有浏览器软件就可以应用系统，从而方便使用者能从不同的地点应用系统。当软件需要升级时，只需要对 Web 服务器端进行改变而无须修改客户端，使得维护工作比较简便。

在涉及复杂的业务处理时，B/S 应用系统需要数据库服务器提供专门的数据服务，这时 B/S 结构可以扩展为 B/S/D 结构。在这种结构中，客户端通过浏览器向 Web 服务器发送请求，Web 服务器上的应用程序向数据库服务器发送请求，数据库服务器将查询结果返回给 Web 服务器，Web 服务器再以 HTML 形式向客户端浏览器返回结果，在浏览器中以页面的形式呈现。

从各方面比较结果来看，校友录网站采用 B/S 更适合。系统是为了充分利用本校的网络资源而建立在广域网上的，面向广大的校友，还要经常进行维护和更新，B/S 结构比 C/S 结构更加适用。

平台采用微软的.NET 平台，.NET 平台由微软公司推出，目前已推出.NET 4.0 版本和 Visual Studio 2010 集成开发环境。.NET 框架提供了丰富的组件，有助于提高软件开发效率，可以容易地生成 ASP.NET Web 应用程序和.NET Web Service。

技术实现选用 ASP.NET，ASP.NET 是微软.NET 框架应用层的一部分，用于开发 B/S 模式的应用程序。ASP.NET 的应用程序可以看成是 HTML+C#（或其他的.NET 编程语言）+Web 控件的组合，使用 ASP.NET 可以快速地形成可发布的 B/S 结构软件产品。

ASP.NET 应用程序运行在 Windows 的 IIS 之上。当一个 HTTP 请求被 IIS 机收到之后，IIS 根据请求为其加载相应的 dll 文件，然后在处理过程中将这条请求发送给 HttpHandler。一个 HTTP 请求有可能经过的 4 条路线，如图 21-9 所示。

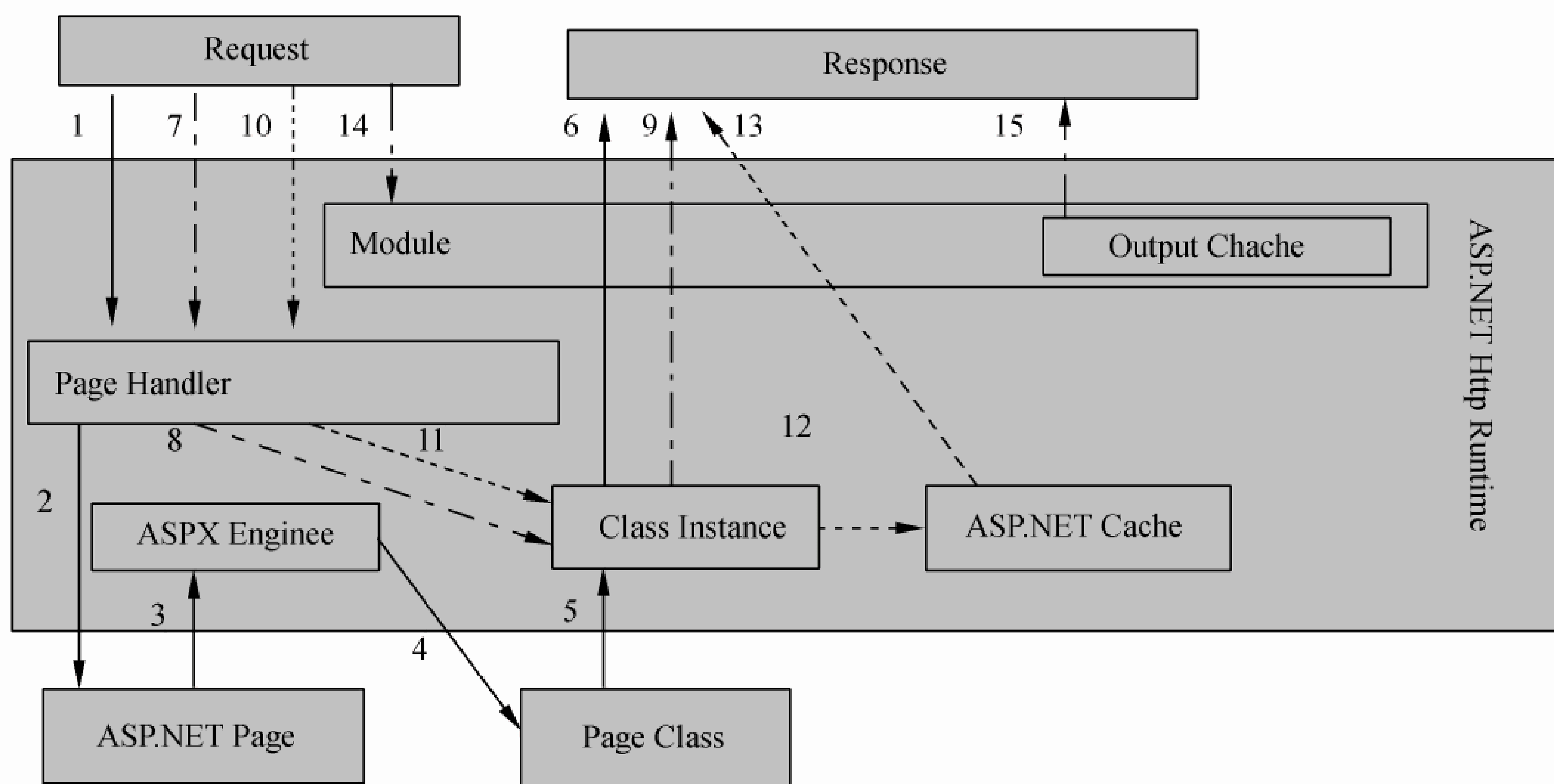


图 21-9　ASP.NET 响应 HTTP 请求路线图

本例采用 Visual Studio 2010 开发工具，使用 B/S 结构，使用 XML 作为基础数据库进行开发。

21.4　数据库设计

数据库是信息系统的核心和基础，把信息系统中大量的数据按一定的模型组织起来，提供存储、维护、检索数据的功能，使信息系统可以方便、及时、准确地从数据库中获得所需的信息。数据库设计是建立数据库及其应用系统的技术，是信息系统开发和建议中的核心技术。由于数据库应用系统的复杂性，为了支持相关程序运行，数据库设计就变得异常复杂，因此最佳设计不可能一蹴而就，而只能是一种“反复探寻，逐步求精”的过程，也就是规划和结构化数据库中的数据对象及这些数据对象之间关系的过程。不论采用什么类型的数据库，设计原理和思想是一致的。

由于本例比较简单，重点不在于系统的功能，而在于实现过程和 XML 的操作，同时管理的对象主要还是客户信息，所以管理的对象也就是一个客户对象了。分析这个对象，管理的属性主要还是客户的基本信息，如姓名、昵称、电话等这些信息。而且使用 XML 作为数据库可以不用理会数据类型和长度问题，只需要按照 XML 的标准格式设计好结点结构即可。设计出的 XML 文件如示例代码 21-1 所示。

示例代码 21-1

```
<?xml version="1.0" encoding="gb2312"?>
<Root>
  <Student Admin="no">
    <Name>张帅</Name>
    <NickName>帅哥</NickName>
    <Pwd>123456</Pwd>
    <Sex>男生</Sex>
    <Birthday>1990-09-09</Birthday>
    <Email>shuaishuai@163.com</Email>
    <QQ>100003</QQ>
    <Msn>shuaishuai@163.com</Msn>
    <Tel>10000000008</Tel>
    <Homepage>http://www.shuaishuaicom</Homepage>
    <Address>北京</Address>
    <Work>待业</Work>
    <Photo>images/张帅.gif</Photo>
    <Time>2008-8-6 20:22:38</Time>
  </Student>
  <Student Admin="no">
...//此处省略部分类似代码
  </Student>
</Root>
```

注意： 设计 XML 文件的时候，需要注意 XML 的特点，比如结点成对出现，结点名区分大小写，根结点只能有一个。另外还需要注意存储格式，尽量存储为 UTF-8 格式，以避免解析不出来或者解析出来是乱码。

上述 XML 文件中有一个根结点，结点名叫 Root 是 XML 格式要求，包含在中间的是子结点，名为 Student 意思是这个结点下面存储的是学生信息。一个 Student 结点代表一个

学生对象，而 Student 结点里面包含的则是管理对象的关键属性，属性值写在结点中间部分。

21.5　系统设计与实现

对于 B/S 程序来说，功能和安全性都是非常值得重视的部分。同时，界面也不能轻视，因为产品给客户的第一印象还是来自界面，如整体页面的布局、色彩的搭配、图片和特效等是否合适，是否符合用户的操作习惯。下面就对整个项目的界面要求和要实现目标进行详细介绍。

21.5.1　界面风格

本系统是基于 B/S 的网络应用程序，从布局角度来说，常见的布局方式有下列几种，如图 21-10 所示。

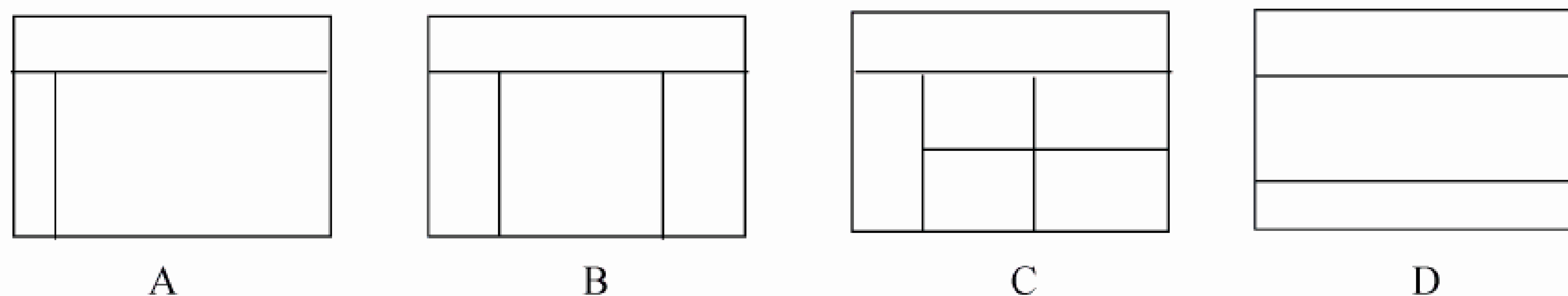


图 21-10　常见的网页布局

其中 A 类常见于应用型系统中，左边为操作菜单，右边为主显示区，如邮箱，OA 等系统，体现了操作集中，主次分明的特点。B 类常见于博客或技术类网站，左右都是文章列表，也可以增加些说明、日历评论等元素进去，中间是正文区，这类布局的特点是重点突出，细节考虑周到。C 类一般见于企业网站和电子商务类，上面是菜单，左边可以是一些子菜单，也可以是导航、调查、公告等元素，右边是分类显示区域，这类网站的特点是功能全面、实用。D 类也常见于博客类站点，上面为主菜单，底部是版权部分，中间就是正文区，这类网站特点是简单，清晰，功能比较单一。本例就采用 D 类布局，原因是没有多少复杂的业务，如果要把校友录功能做全面，商业化运营的话，一般会选择 A 类布局。

布局确定之后，就需要分析系统的风格和配色了，可以根据客户的要求给客户一些建议或者选择，让客户决定。不管采用哪种方式，系统的风格一旦确定，则整个站点都必须一致，不能一个页面一个风格。拿新浪网为例，整个站点的基本色为橘黄色，整个站点的布局基本为图 21-10 的 D 类更细些。在新浪访问任何一个页面，发现它的风格总是统一的，让人感觉不到变化，也就是说，感觉总是熟悉的，这就是为什么要保持风格统一了。风格的统一包含布局和配色，意思是一个网站有变化的部分也有不变化的部分，本例中的菜单区和版权区就属于不变的部分，即每个页面都有并且一模一样，要实现这个目标大可不必一个一个页面去画，只需要采用 ASP.NET 特有的“母版页”技术即可实现。

使用 ASP.NET 母版页可以为应用程序中的页创建一致的布局。单个母版页可以为应用程序中的所有页（或一组页）定义所需的外观和标准行为。然后可以创建包含要显示的

内容的各个内容页。当用户请求内容页时，这些内容页与母版页合并以将母版页的布局与内容页的内容组合在一起输出。运行原理如图 21-11 所示。

从图 21-11 中可以看出，母版页和模板类似，都是提供了一个统一的风格布局实现，每个内容页只需要关注自己的实现部分，运行的时候会被合并成为一个页面，意思是把内容页合并到母版页中的占位符处。实现方式是在 VS 资源管理器的网站上右击，在弹出的快捷菜单中选择“新建项”，创建一个母版页。布局编辑完成后，在母版页上右击，在弹出的快捷菜单中选择“新建内容页”即可。

配色采用有利于眼睛健康的绿色为基准色调，配以蓝色，会让客户感觉到惬意并且不会疲倦。

21.5.2　系统功能模块设计

根据前面的需求分析和主要功能说明，设计系统时就需要分模块设计，方便开发，方便组织。本例主要包含以下模块，如图 21-12 所示。

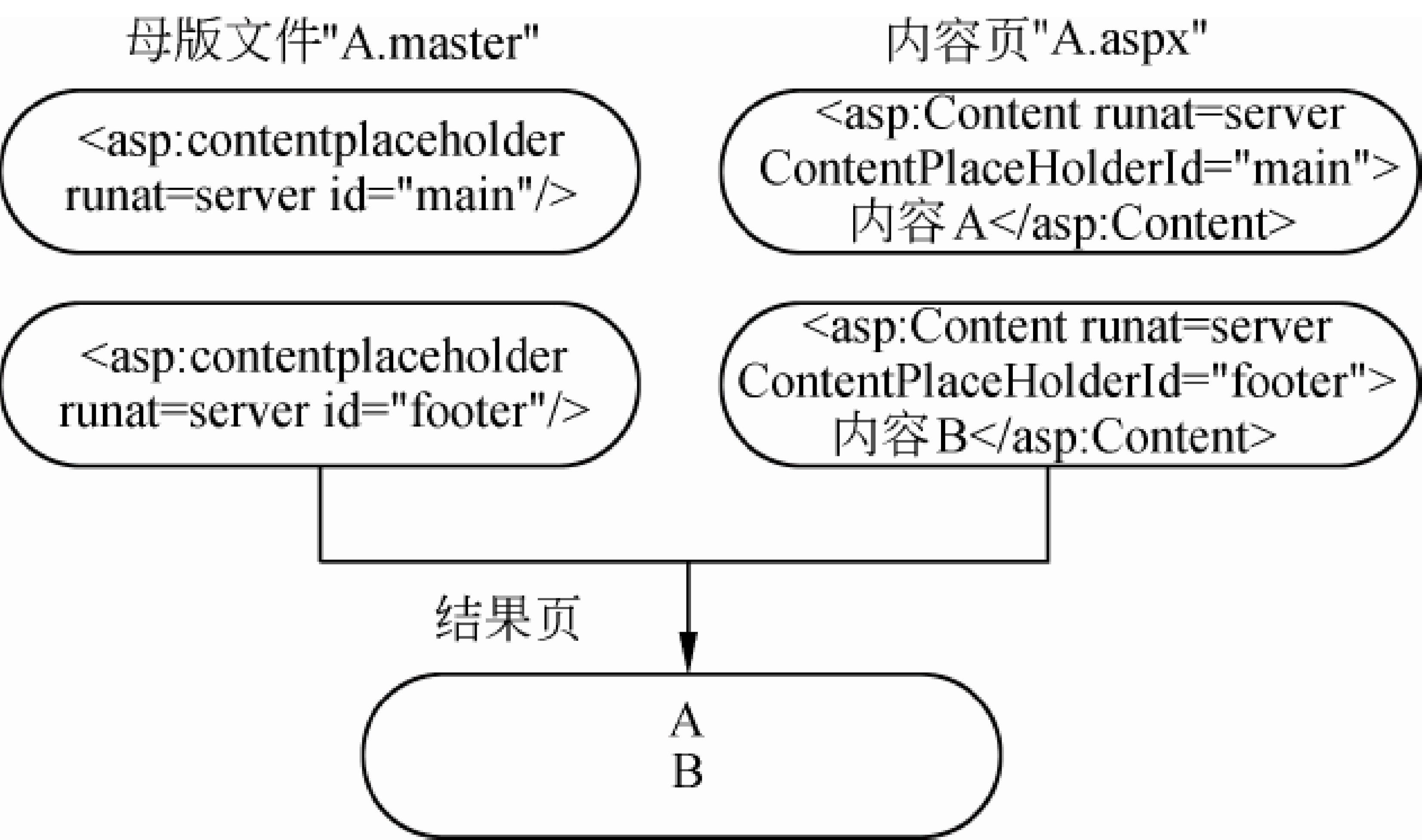


图 21-11　母版页运行原理

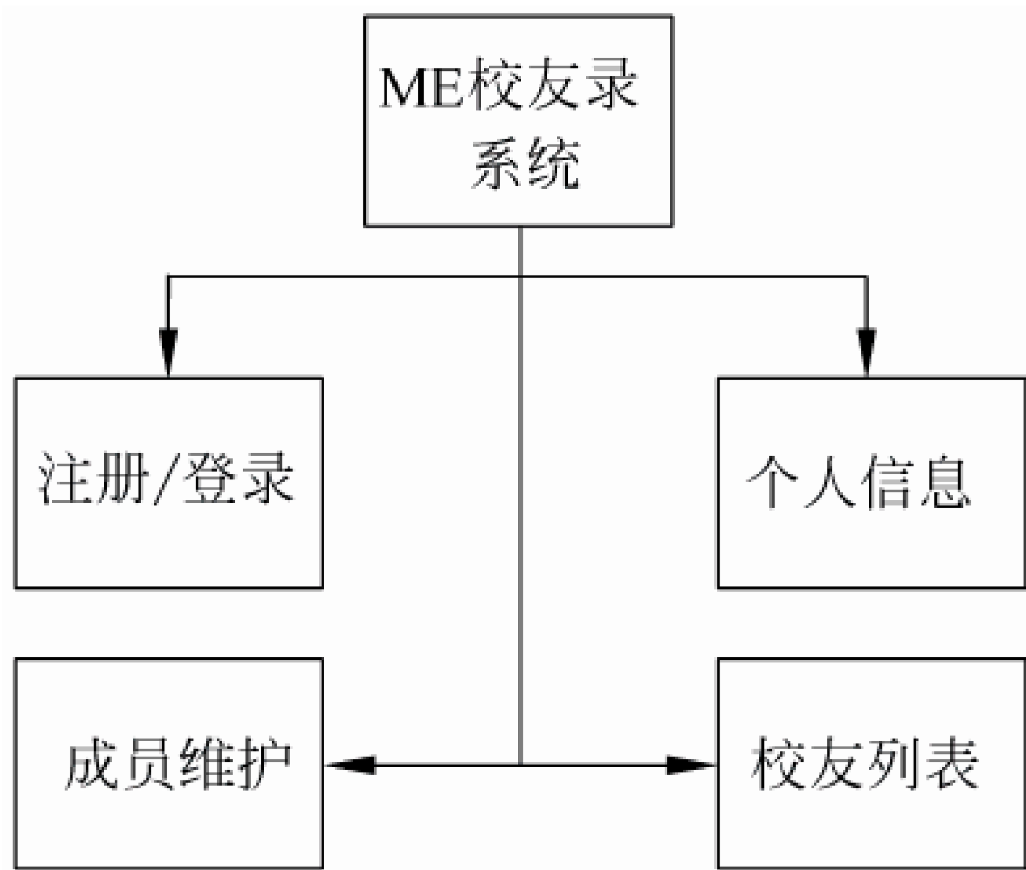


图 21-12　系统模块

21.5.3　系统通用类实现

由于本次案例采用的数据库是 XML，必然会涉及 XML 的读写和查询操作，并且这些操作是通用的，所以可以先写成一个通用的操作类，以便在后续的业务中灵活调用。这个操作类应该具备的功能如表 21-2 所示。

表 21-2　XML操作类功能

名　　称	类　　型	摘　　要
DeleteNode	bool	删除一个结点
GetDs	System.Data.DataSet	根据 XML 文件的结点路径，返回一个 DataSet 数据集
GetXmlFilePath	string	返回 XML 文件实际路径
InsertAttrib	bool	添加属性
InsertNode	bool	插入一个结点，带多个子结点
NodeCount	int	获取子结点个数
Save	void	保存文件

续表

名　　称	类　　型	摘　　要
SelectAttrib	string	属性查询，返回属性值
SelectNode	bool	结点值查询判断
SelectNodeText	string	结点查询，返回结点值
UpdateAttrib	bool	修改属性
UpdateNode	bool	更新多个结点值
UpdateNode	bool	更新一个结点的内容
XMLOperater		构造函数，导入 XML 文件

按照此功能表设计出的类图，如图 21-13 所示。

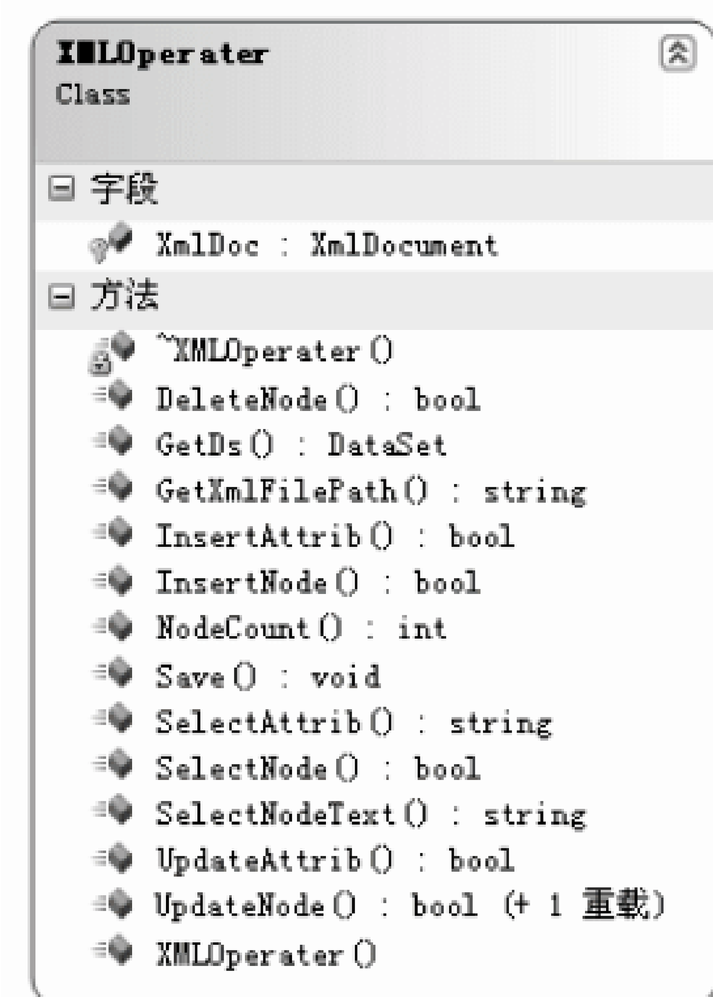


图 21-13　XML 操作类图

（1）GetXmlFilePath()方法：用于返回 XML 文件实际路径，由于 web.config 中存储的是相对路径，而在对 XML 操作的时候必须按照实际路径，所以需要方法获取。获取原理是使用 Server.MapPath()方法根据虚拟路径转换对应的物理路径，转换代码如示例代码 21-2 所示。

示例代码 21-2

```
/// <summary>
/// 返回 XML 文件实际路径
/// </summary>
/// <param name="xmlFile">文件虚拟路径</param>
public string GetXmlFilePath(string xmlFile)
{
    return HttpContext.Current.Server.MapPath(xmlFile);
}
```

（2）DeleteNode()方法：原理是根据参数传递的结点名，调用 XmlDocument 对象的 SelectSingleNode 方法找到第一个匹配的结点。然后返回父结点，调用 RemoveChild()方法删除父结点下的所有子结点，包括自己，实现结点的删除操作。代码如示例代码 21-3 所示。

示例代码 21-3

```
/// <summary>
/// 删除一个结点
/// </summary>
/// <param name="Node">要删除的结点</param>
public bool DeleteNode(string Node)
{
    try
    {
        //调用 XmlNode 的 RemoveChild()方法来删除结点及其所有子结点
        XmlDoc.SelectSingleNode(Node).ParentNode.RemoveChild(XmlDoc.
        SelectSingleNode(Node));
        return true;
    }
    catch
    {
        return false;
    }
}
```

（3）GetDs()方法：这个方法是将 XML 的数据和架构信息读到临时数据集中，以备使用。代码如示例代码 21-4 所示。

示例代码 21-4

```
/// <summary>
/// 根据 Xml 文件的结点路径，返回一个 DataSet 数据集
/// </summary>
/// <param name="XmlPathNode">Xml 文件的某个结点</param>
public DataSet GetDs(string XmlPathNode)
{
    DataSet ds = new DataSet();
    try
    {
        System.IO.StringReader read = new System.IO.StringReader(XmlDoc.
        SelectSingleNode(XmlPathNode).OuterXml);
        ds.ReadXml(read);
              //利用 DataSet 的 ReadXml 方法读取 StringReader 文件流
        read.Close();
    }
    catch
    { }
    return ds;
}
```

（4）InsertAttrib()方法：为指定的结点增加属性名和属性值，思路同样是先找到要加的结点，然后调用 XmlElement 对象的 SetAttribute()方法增加信息。代码如示例代码 21-5 所示。

示例代码 21-5

```
/// <summary>
/// 添加属性
/// </summary>
/// <param name="MainNode">属性所在结点</param>
/// <param name="Attrib">属性名</param>
/// <param name="AttribContent">属性值</param>
public  bool  InsertAttrib(string  MainNode,  string  Attrib,  string
AttribContent)
{
    try
    {
        XmlElement objNode = (XmlElement)XmlDoc.SelectSingleNode(MainNode);
                                    //强制转化成 XmlElement 对象
        objNode.SetAttribute(Attrib, AttribContent);
                                    //XmlElement 对象添加属性方法
        return true;
    }
    catch
    {
        return false;
    }
}
```

（5）InsertNode()方法：为一个指定的结点插入多个子结点。实际上是调用 XmlElement 对象的 AppendChild()方法实现插入结点操作。代码如示例代码 21-6 所示。

示例代码 21-6

```
/// <summary>
/// 插入一个结点，带 N 个子结点
/// </summary>
/// <param name="MainNode">插入结点的父结点</param>
/// <param name="ChildNode">插入结点的元素名</param>
/// <param name="Element">插入结点的子结点名数组</param>
/// <param name="Content">插入结点的子结点内容数组</param>
public bool InsertNode(string MainNode, string ChildNode, string[]Element,
string[] Content)
{
    try
    {
        XmlNode objRootNode = XmlDoc.SelectSingleNode(MainNode);
                                                    //声明 XmlNode 对象
        XmlElement objChildNode = XmlDoc.CreateElement(ChildNode);
                                                    //创建 XmlElement 对象
        objRootNode.AppendChild(objChildNode);
        for (int i = 0; i < Element.Length; i++)     //循环插入结点元素
        {
            XmlElement objElement = XmlDoc.CreateElement(Element[i]);
            objElement.InnerText = Content[i];
            objChildNode.AppendChild(objElement);
        }
        return true;
    }
    catch
    {
        return false;
    }
}
```

（6）NodeCount()方法：获取指定结点的子结点个数，思路是找到该结点返回 ChildNodes 的 Count 属性值即可。代码如示例代码 21-7 所示。

示例代码 21-7

```
/// <summary>
/// 获取子结点个数
/// </summary>
/// <param name="XmlPathNode">父结点</param>
public int NodeCount(string XmlPathNode)
{
    int i = 0;
    try
    {
        i = XmlDoc.SelectSingleNode(XmlPathNode).ChildNodes.Count;
    }
    catch
    {
        i = 0;
    }
    return i;
}
```

（7）SelectAttrib()方法：根据结点名和属性名获取对应的属性值。本方法主要用到结点的 InnerText 属性，主要是获取结点值用。代码如示例代码 21-8 所示。

示例代码 21-8

```
/// <summary>
/// 结点查询，返回结点值
/// </summary>
/// <param name="XmlPathNode">结点的路径</param>
public string SelectNodeText(string XmlPathNode)
{
    string  nodeTxt = XmlDoc.SelectSingleNode(XmlPathNode).InnerText;
    if ( nodeTxt == null ||  nodeTxt == "")
        return "";
    else
        return  nodeTxt;
}
```

（8）UpdateAttrib()方法：更新结点属性。根据参数传入的结点名找到该结点，再根据原属性名找到该属性，然后对它的 value 属性赋新属性值。代码如示例代码 21-9 所示。

示例代码 21-9

```
/// <summary>
/// 修改属性
/// </summary>
/// <param name="XmlPathNode">属性所在的结点</param>
/// <param name="Attrib">属性名</param>
/// <param name="Content">属性值</param>
 public bool UpdateAttrib(string XmlPathNode, string Attrib, string
 AttribContent)
{
    try
    {
        //修改属性值
        XmlDoc.SelectSingleNode(XmlPathNode).Attributes[Attrib]. Value =
        AttribContent;
        return true;
    }
    catch
    {
        return false;
    }
}
```

（9）UpdateNode()方法：有 2 个版本，2 个参数的是更新一个结点的值，3 个参数的是更新多个结点的值。更新方式相同，都是找到该结点，为结点的 InnerText 属性赋新值，不同的是重载版本需要循环赋值。代码如示例代码 21-10 所示。

示例代码 21-10

```
/// <summary>
/// 更新 N 个结点值
/// </summary>
/// <param name="XmlParentNode">父结点</param>
/// <param name="XmlNode">子结点</param>
/// <param name="NodeContent">子结点内容</param>
public bool UpdateNode(string XmlParentNode, string[] XmlNode, string[]
NodeContent)
{
```

```
    try
    {
        //根据结点数组循环修改结点值
        for (int i = 0; i < XmlNode.Length; i++)
        {
            XmlDoc.SelectSingleNode(XmlParentNode + "/" + XmlNode[i]). Inner
            Text = NodeContent[i];
        }
        return true;
    }
    catch
    {
        return false;
    }
}
```

21.5.4　注册功能实现

用户登录网站后的第一件事情就是注册，需要填写用户信息，例如用户名、密码、真实姓名、生日、邮箱等信息，其中一些为选填项，用户还可以上传个人头像。如果用户正确输入信息，确认无误后，提交之后注册成功。这里有必要对用户的输入数据合法性进行验证，常见的验证目标有：非空、相等、长度和特定规则等。本例使用微软自带的几个验证空间，如 RequiredFieldValidator（必填项）来规范用户输入，RegularExpressionValidator（模式匹配）来检查项与正则表达式定义的模式是否匹配，使用 CompareValidator，检查用户输入的密码是否一致。如果输入错误，则显示错误信息。例如用户输入的真实姓名是否为中文，E-mail 地址是否合法，电话号码格式是否正确等。

如果用户输入已经存在的用户名，则系统提示已经存在此用户，如果系统中一个成员都没有则不需要判断，实现代码如示例代码 21-11 所示。

示例代码 21-11

```
if (!IsPostBack)
     {
         XMLOperater op = new XMLOperater(xmlFile);
                                   //创建 XmlOp 类对象
         if (op.NodeCount("//Root") == 0)
                                   //如果一个同学成员都没有，则取消姓名验证
             this.CustomValidator1.Enabled = false;
     }
```

在通过所有验证的情况下，还要判断是否上传头像和头像图片的格式，实现代码如示例代码 21-12 所示。

示例代码 21-12

```
//提交注册信息
protected void btnReg_Click(object sender, EventArgs e)
{
    if (IsValid)    //如果通过所有验证
    {
        string _name = txtUserName.Text.Trim();
```

```
    string  nickName = txtNickName.Text.Trim();
    string _pwd = txtUserPwd.Text.Trim();
    string  sex = rbtnMan.Checked ? rbtnMan.Text : rbtnWoman.Text;
                                //判断性别
    string  birthday = txtBirthday.Text.Trim();
    string _email = txtEmail.Text.Trim();
    string  qq = txtQQ.Text.Trim();
    string  msn = txtMsn.Text.Trim();
    string  Tel = txtTel.Text.Trim();
    string _address = txtAddress.Text.Trim();
    string  work = txtWork.Text.Trim();
    string  homepage = txtHomePage.Text.Trim();
    string  photo = "";
    string _time = DateTime.Now.ToString();
    //上传图片
    string  file = this.FileUpload1.FileName.ToString().
    ToLower();
    if ( file != "")
    {
        string  fileType = this.FileUpload1.PostedFile.
        ContentType;
        if ( fileType.Substring(0, 5) == "image")
                                //判断是否图片文件
        {
             photo = "images/" +  name +  file.Substring( file.
            LastIndexOf("."));
            this.FileUpload1.SaveAs(Server.MapPath( photo));
                                //保存图片
        }
        else
        {
            manager.Msg("图片格式不对!");
            return;             //终止执行
        }
    }
    //结点数组
    string[] stu ={"Name", "NickName", "Pwd", "Sex", "Birthday","Email",
    "QQ", "Msn", "Tel", "Homepage", "Address", "Work", "Photo","Time" };
    //结点值数组
    string[] stuInfo ={ name,  nickName,  pwd,  sex,  birthday,  email,
    _qq, _msn, _Tel, _homepage, _address, _work, _photo,_time };
    XMLOperater op = new XMLOperater(xmlFile);
    //调用 XML 操作类的 InsertNode()方法, 插入新的同学成员记录
    if (op.InsertNode("//Root", "Student", stu, stuInfo))
    {
        op.InsertAttrib("//Root/Student[Name='" +  name + "']", "Admin",
        "no");
        op.Save(xmlFile);           //保存 XML 文档
        manager.Msg("恭喜,已注册成功!");
        ClearTextBox();             //清空所有文本框
    }
}
```

最后构建好数组，调用 XML 操作类的增加结点方法，实现数据插入并且保存 XML 文件，清空界面的文本框。

21.5.5　登录功能

注册完毕后，就可用刚才注册的用户名和密码进行登录，使用系统提供的服务了。登录，其实是在验证用户的身份，所以只需要核对好用户名和密码的合法性即可。

注意：在登录之前为了用户体验，一般都需要预先检测该用户是否已经登录过。如果是已经登录过的用户，则不需要再次进行验证，直接进入。如果是新用户则需要登录，根据 Cookies 来判断用户状态。当然，在登录成功之后需要把用户的状态存储起来，方便下次判断。

登录前判断代码如示例代码 21-13 所示。

示例代码 21-13

```
protected void Page Load(object sender, EventArgs e)
       {
           if (!IsPostBack)
           {
               //读取 Cookie 里的账号和密码
               HttpCookie cookie = Request.Cookies["userInfo"];
               if (cookie != null && cookie["userName"].ToString() != "")
               {
                   this.userLogin.InnerHtml = "<p>" + cookie.Values ["user-
                   Name"].ToString() + " 同学，欢迎你回来 !<br/><br/><a href=
                   'MyInfo.aspx'>进入我的同学录</a></p>";
               }
           }
       }
```

登录验证方法代码如示例代码 21-14 所示。

示例代码 21-14

```
/// <summary>
/// 同学录成员登录验证，验证成功后把信息写入 Cookie
/// </summary>
/// <param name="userName">姓名</param>
/// <param name="userPwd">密码</param>
public bool UserLogin(string userName, string userPwd, int cookieDay)
{
    XMLOperater op = new XMLOperater(xmlFile);
    if (op.SelectNode("//Root/Student", 0, userName) && op.SelectNode
    ("//Root/Student", 2, userPwd))
    {
        HttpCookie cookie = new HttpCookie("userInfo");
                                                //创建 Cookie 对象
        cookie["userName"] = userName;
        cookie["userPwd"] = userPwd;
        cookie.Expires = DateTime.Now.AddDays(cookieDay);
        HttpContext.Current.Response.Cookies.Add(cookie);
                                                //写入 Cookie 值
        return true;
    }
```

```
    else
    {
        return false;
    }
}
```

注意：使用 Cookies 是一个比较危险的方式，原因是它是把信息存储在用户本地的机器上，有导致身份信息泄露的风险。并且如果不是家庭用户，记忆用户登录信息也是比较危险的操作方式。一般项目中会提示是否保存 Cookies 信息，并且同时会把记录和不记录的区别说得很清楚。

21.5.6　详细信息

详细信息的显示包含显示自己和显示校友的详细信息，需要显示姓名、昵称、生日包括头像等。为了方便显示，本例采用 Repeater 数据绑定控件进行展示。Repeater 是高效的数据展示控件，可以做到精确控制，不会生成额外代码，使用的时候只需要提供一个数据源给它即可。后台绑定代码如示例代码 21-15 所示。

示例代码 21-15

```
//数据绑定到 Repeater
private void BindToRepeater(Repeater rp)
{
    DataSet ds = new DataSet();
    XMLOperater op = new XMLOperater(ConfigurationManager.AppSettings
     ["xmlFile"]);
    ds = op.GetDs("//Root/Student[Name='" +  userName + "']");
    rp.DataSource = ds.Tables[0].DefaultView;
    rp.DataBind();
}
```

当然在前台需要展示的地方需要指定显示字段，可以有很多种展示方式，常用的有 Eval("字段名")，Bind("字段名")，本例采用的是：

```
姓名：<%# DataBinder.Eval(Container, "DataItem.Name") %><br />
```

注意：数据展示控件的字段绑定两种方式的区别在于 Eval 是单向只读的，而 Bind 是双向的，如内存或界面有一方修改，则两边都会被修改。

21.5.7　修改我的信息

修改信息的原理和上述绑定类似，多出的步骤是把新修改的数据获取和保存到 XML 中，所以本节的重点不在于显示，要做的是获取和保存。

在提交修改之前和注册一样都需要验证数据的合法性，所以获取数据保存之前需要判断页面的合法性，然后获取，获取修改数据代码如示例代码 21-16 所示。

示例代码 21-16

```
string  nickName = ((TextBox)this.rpt myInfo.Items[0].FindControl ("txt-
NickName")).Text.Trim();
string  pwd = ((TextBox)this.rpt myInfo.Items[0].FindControl("txtPwd")).
Text.Trim();
string  birthday = ((TextBox)this.rpt myInfo.Items[0].FindControl
("txtBirthday")).Text.Trim();
…//省略部分类似代码
```

获取到新数据之后就可以执行保存操作，但是需要注意，如果修改了头像，就需要再次上传头像，检测及保存代码如示例代码 21-17 所示。

示例代码 21-17

```
XMLOperater op = new XMLOperater(ConfigurationManager. AppSettings["xml-
File"]);
string _type = _photoType;
string  file =  photo.ToLower();
if (_photo != "")           //如果没有选择图片时，就不需修改记录中的 Photo 属性
{
    if ( photoType.Substring(0, 5) == "image")
    {
        //上传图片
         photo = "images/" +  userName +  file.Substring
       ( file.LastIndexOf ("."));
        ((FileUpload)this.rpt myInfo.Items[0]. FindControl ("FileUploadIm-
        age")).SaveAs(Server.MapPath( photo));
    }
    else
    {
        manager.Msg("图片格式不对!");
        return;
    }
    string _xmlParentNode = "//Root/Student[Name='" + _userName + "']";
    string[]  xmlNode = { "NickName", "Pwd", "Birthday", "QQ", "Msn", "Tel",
    "Email", "Address", "Work", "Homepage", "Photo", "Time", };
    string[]  xmlNodeTxt = {  nickName,  pwd,  birthday,  qq,  msn,  Tel,
     Email,  address,  work,  homepage,  photo,  time };
    op.UpdateNode( xmlParentNode,  xmlNode,  xmlNodeTxt);
}
else  //选择了图片，需要修改记录中的 Photo 属性
{
    string  xmlParentNode = "//Root/Student[Name='" +  userName + "']";
    string[]  xmlNode = { "NickName", "Pwd", "Birthday", "QQ", "Msn", "Tel",
    "Email", "Address", "Work","Homepage", "Time", };
    string[] _xmlNodeTxt = { _nickName, _pwd, _birthday, _qq, _msn, _Tel,
      Email,  address,  work,  homepage,  time };
    op.UpdateNode( xmlParentNode,  xmlNode,  xmlNodeTxt);
}
op.Save(ConfigurationManager.AppSettings["xmlFile"]);  //保存新的信息
```

21.5.8　校友列表信息

作为校友录的最关键点，就是查看校友的信息，可以先在校友列表中选择需要查看的校友名字，单击进入详细信息查看页面。需要把校友大体信息以列表的形式显示出来，本例选用 ASP.NET 下面的 GridView 数据列表控件。绑定代码如示例代码 21-18 所示。

示例代码 21-18

```
//数据绑定到 DataGrid，显示班级所有成员
private void BindToDataGrid()
{
    try
    {
        XMLOperater op = new XMLOperater(_xmlFile);//声明 XmlOp 类对象
        ds = op.GetDs("//Root");
        this.gv list.DataSource = ds;
        this.gv list.DataBind();
        Session["thisDS"] = ds;             //把数据集保存到 Session 变量中
    }
    catch
    {
        Response.Write("出现不明错误 !");
        Response.End();
    }
}
```

这里把校友列表信息组成的 DataSet 暂存起来，是为了下面校友搜索功能的简化，在输入校友名字的关键字之后不需要再次读取 XML 文件，提高系统运行效率。

但是同时由于姓名是采用汉字绑定和当作参数传递的，就涉及了一个传输编码解码的问题。如果不对 URL 方式传递汉字参数进行编码和解码，那么接收到的将会是乱码，导致系统不能正常工作，所以在传递前需要编码，编码用到 Server 对象的编码方法 UrlEncode()，编码方法使用示例如示例代码 21-19 所示。

示例代码 21-19

```
/// <summary>
/// 对使用 URL 方式传递的汉字采用编码，防止乱码
/// </summary>
public string GetEncode(string str)
{
    return Server.UrlEncode(str);
}
```

然后在界面里面调用这个编码方法：

```
<a href='stuInfo.aspx?id=<%#GetEncode(Eval("Name").ToString()) %>'><%#
Eval ("Name") %></a>
```

就可以在地址栏看到类似“=%B9%FE%B9%FE&f=8&wd=%B9%FE%B9%FE%CF%ED%D7%EE%D0%C2%D7%EE”说明编码成功。之后需要在接收方接收到这串“乱码”后解码，用的同样是 Server 对象的 UrlDecode()方法，使用方法如下：

```
_name =Server.UrlDecode(Request["id"]);   //获取传递的参数
```

然后在接收方按照解码后的参数查询相应的内容显示即可。

21.5.9　成员管理

成员管理就是设定管理员，如果已经是管理员则显示“取消管理”，还有删除功能，这个页面只允许系统管理员操作。系统管理员整个系统只有一个，其用户名和密码是保存在 web.config 中的，增加了一个配置节，将验证模式改为 Forms，配置方式如示例代码 21-20 所示。

示例代码 21-20

```
<authentication mode="Forms">
  <forms name="mytxl" loginUrl="Default.aspx" protection="All">
    <credentials passwordFormat="MD5">
      <user name="admin" password="21232f297a57a5a743894a0e4a801fc3"/>
    </credentials>
  </forms>
```

为了安全起见，密码采用不可逆加密算法 MD5，MD5 的使用方式如下：

```
System.Web.Security.FormsAuthentication.HashPasswordForStoringInConfigF-
ile("admin","md5")
```

把要加密的字符串当作第一个参数传递进去，第二个参数是加密方式，取值可以是 md5，也可以是 SHA1。

在进行成员管理之前需要进行管理员身份确认，确认实质上就是将用户输入的用户名和密码与配置文件中的进行比较，验证采用的是 ASP.NET 提供的验证方法 Authenticate(用户名，密码)，具体代码如示例代码 21-21 所示。

示例代码 21-21

```
/// <summary>
/// 超级管理员登录身份验证，登录成功后把用户信息写入 Session
/// </summary>
/// <param name="admin">账号</param>
/// <param name="pwd">密码</param>
public bool SuperAdmin(string admin, string pwd)
{
    //利用 Web.Security 类下的验证方法验证身份
    if (System.Web.Security.FormsAuthentication.Authenticate(admin, pwd))
    {
        HttpContext.Current.Session["SuperAdmin"] = "yes";  //超级管理员
        HttpContext.Current.Session["NormalAdmin"] = "yes"; //一般管理员
        return true;
    }
    else
    {
        return false;
    }
}
```

一旦验证成功将会在 Session 中存储一个标识值，然后跳转到操作页面。后续操作就根据 Session 中存储的值进行判断。

在管理界面涉及一个问题：如果该用户原来就是管理员，则不能显示“设置为管理”，而应该显示“取消管理”。反之一个用户原来不是管理员，就不能显示“取消管理”，这个判定工作需要在列表绑定的时候进行控制。一般类似这样的控制是加在 GridView 的 ItemDataBound 事件下，判断代码如示例代码 21-22 所示。

示例代码 21-22

```
if (e.Item.ItemType == ListItemType.AlternatingItem || e.Item.ItemType ==
ListItemType.Item)
{
    ((LinkButton)(e.Item.Cells[6].Controls[0])).Attributes.Add("onclick",
    "return confirm('你确定要删除吗？');");
     if (Convert.ToString(Session["SuperAdmin"]) == "yes") //若为超级管理员
      {
         ((Button)(e.Item.Cells[7].FindControl("btnSetAdmin"))).Visible =
          true;                                           //显示设置管理员按钮
          ((Button)(e.Item.Cells[7].FindControl("btnNoAdmin"))).Visible =
           true;                                          //显示取消管理员按钮
     }
}
```

当然，在界面绑定的按钮是出现在列表中的，需要把按钮所在列的主键和命令传递到后台，后台才可以正确执行任务，按钮的绑定需要增加一个属性，以便记录命令名和主键值。如：

```
<asp:Button ID="btnSetAdmin" runat="server" CommandName="SetAdmin" Text="
设为管理员" Visible="false" />
```

其中的 CommandName 就是命令名，在后台代码中获取主键进行相关操作，代码如示例代码 21-23 所示。

示例代码 21-23

```
//一般管理员的设置
protected void DataGrid1 ItemCommand(object source, DataGridComm-
andEventArgs e)
{
    AlumniManager manager = new AlumniManager();
    XMLOperater op = new XMLOperater(xmlFile);
    //设置一般管理员
    if (e.CommandName == "SetAdmin")
    {
        string _adminName = this.DataGrid1.DataKeys[e.Item. ItemIndex].
        ToString();
        if (op.UpdateAttrib("//Root/Student[Name='" + _adminName + "']",
        "Admin", "yes"))
        {
            op.Save(xmlFile);
            manager.Msg("设置成功!");
            BindToDataGrid();
        }
    }
```

```
    //取消一般管理员
    if (e.CommandName == "NoAdmin")
    {
        string _adminName = this.DataGrid1.DataKeys[e.Item. ItemIndex].
        ToString();
        if (op.UpdateAttrib("//Root/Student[Name='" + _adminName + "']",
        "Admin", "no"))
        {
            op.Save(xmlFile);
            manager.Msg("设置成功!");
            BindToDataGrid();
        }
    }
}
```

其中的 BindToDataGrid()方法是绑定列表方法，前面已经重复多次。成员管理还有一个很重要的操作就是删除用户，删除分两步操作：删除数据记录和头像图片，需要注意的是在删除头像图片之前需要判断图像是否存在，否则会引发异常，最后进行刷新页面操作即可，对应代码如示例代码 21-24 所示。

示例代码 21-24

```
//删除某个同学
protected void DataGrid1_DeleteCommand(object source, DataGridComm-
andEventArgs e)
{
    XMLOperater op = new XMLOperater(xmlFile);
    //取得关键字段
    string  delName = this.DataGrid1.DataKeys[e.Item.ItemIndex]. ToStr-
    ing();
    string  delNode = "//Root/Student[Name='" +  delName + "']";
    string  photo = op.SelectNodeText("//Root/Student[Name='" +  delName +
    "']/Photo");
    if (op.DeleteNode(_delNode))    //删除操作
    {
        op.Save(xmlFile); //保存 XML 文档
        if (System.IO.File.Exists(HttpContext.Current.Server. MapPath( ph-
        oto)))
        {
            try
            {
                //删除相应图片
                System.IO.File.Delete(HttpContext.Current.Server. MapPath
                ( photo));
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }
        Response.AddHeader("refresh", "0"); //刷新整个页面
    }
}
```

注意最后一句 Response.AddHeader("refresh", "0");的意思是刷新整个页面，最终体现成 HTML 代码是：

```
//就是在 HTML 文件中的 <head> 部分加上这一句：
<meta  http-equiv= "refresh " content= "0;url= 'http://www.youiname.net
' ">
```

起到客户端强制刷新跳转的作用，这里也要提醒读者这样的做法对于网站创办者来说初衷是好的，但是刷新时间设置过短会被搜索引擎降权。

21.5.10　整站登录验证

基于 B/S 的系统不像 WinForm 程序，身份验证把握好一个入口就可以达到目的，每一个网页都是有地址的，用户完全可以不通过超级连接而在浏览器地址栏直接输入网址进入。所以对于网页程序身份验证就是一个比较麻烦的事情，每个页面都验证明显不合适，还可能出现安全漏洞。本例采用的是基类页验证方式，用的思想是面向对象的继承，基本原理如图 21-14 所示。

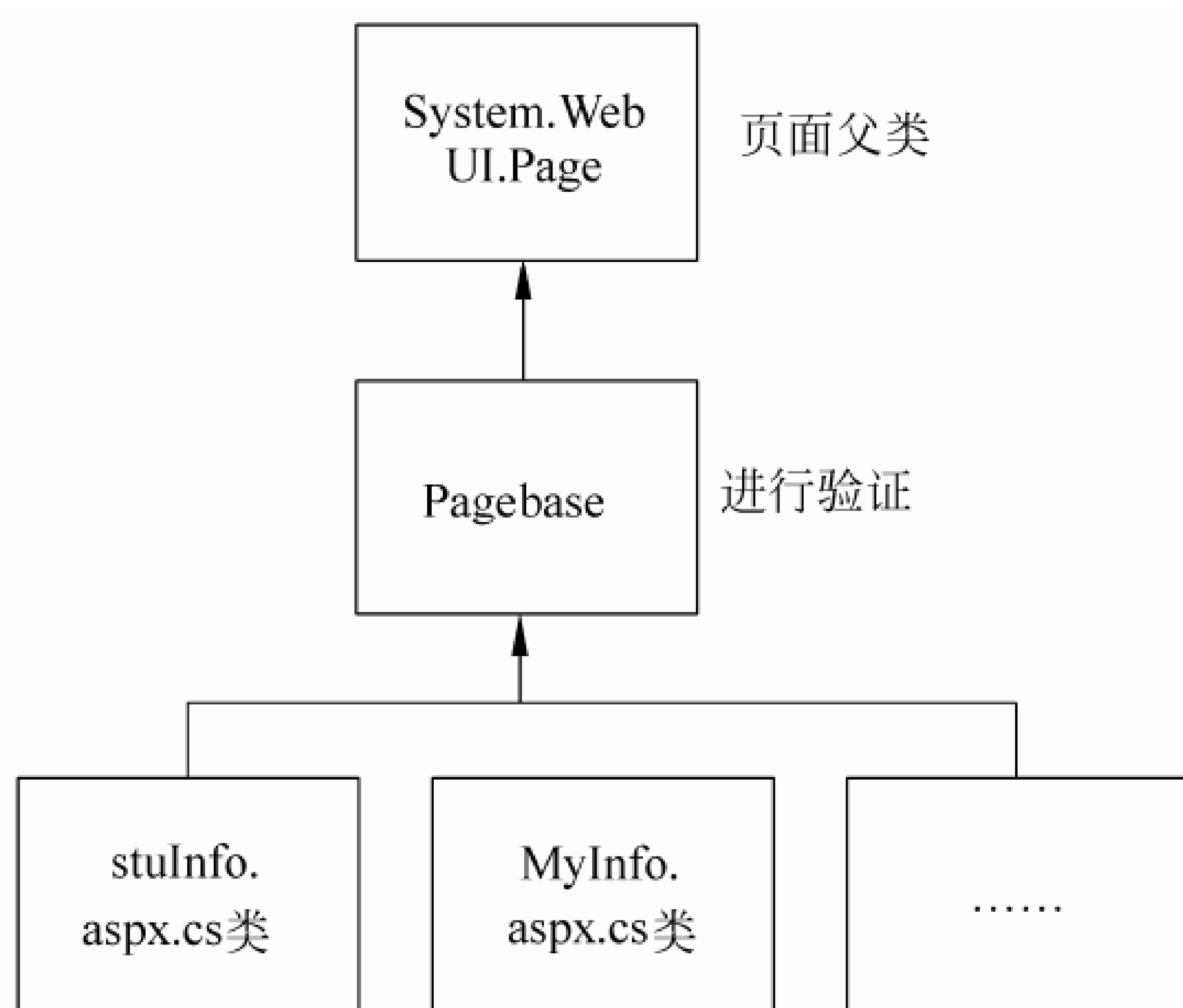


图 21-14　页面类继承关系图

在 PageBase 类中重写 Page 类的 OnLoad()方法，并在 OnLoad 下进行身份验证。如果通过则继续访问，如果不通过，则导航到登录页面，然后让需要验证的页面类继承自 PageBase 类即可实现调用，代码如示例代码 21-25 所示。

示例代码 21-25

```
/// <summary>
/// 重写 Load()方法
/// </summary>
protected override void OnLoad(EventArgs e)
{
    string _userName = null;  //记录用户名变量
    string _userPwd = null;   //记录用户密码变量
    //读取 Cookie 里的账号和密码并赋值给变量
    HttpCookie cookie = Request.Cookies["userInfo"];
    if (cookie != null)
    {
        _userName = cookie.Values["userName"].ToString();
        _userPwd = cookie.Values["userPwd"].ToString();
```

```
    }
    //页面第一次加载
    if (!IsPostBack)
    {
        if (_userName == null || _userName == "") //非法访问时
        {
            Response.Write("你没有登录，<a href='default.aspx'>请登录</a>!");
            Response.End();
        }
    }
    base.OnLoad(e);
}
```

这样当用户没有经过身份验证而要登录时，就会被跳转到登录页面，从而完成所有页面的验证操作。

21.6　项 目 小 结

本系统基本实现了校友录管理系统的主要功能，实现了用户的登录和注册、个人信息修改、图片上传、班级成员列表、管理员管理等模块。采用面向对象的软件工程方法，对系统进行了分析和设计，基本上实现了系统的基本功能。经过系统的部署和对系统进行测试，初步运行后，系统达到了预期的目标，但是还有很多改进的地方需要改进和完善。

由于时间和项目本身意义的原因，本例没有把重点放在具体校友录细节功能的实现，而是放在设计开发过程和网络开发经验上，所以功能比较单一。另外，也没有运用到系统的架构知识（可以参考宾馆管理系统），但是对 XML 的操作及实战应该有一个比较清晰的阐述了。